

Design and Development of support for GPU Unified Memory in OMPSS

Master in Innovation and Research in Informatics (MIRI)

High Performance Computing (HPC)

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC)

Aimar Rodriguez Soto
aimar.rodriguez@bsc.es

Supervisor: Vicenç Beltran Querol
Cosupervisor: Antonio Jose Peña Monferrer

31 of October, 2017

Abstract

Heterogeneous computing has become prevalent as part of High Performance Computing in the last decade, with asynchronous devices such as Graphics Processing Units constantly evolving. As HPC becomes more specialized and heterogeneous devices become more advanced and implement new features, a challenge is presented to traditional programming models.

Programming models and tools needs to adapt in order to keep a competitive performance. Due to this, a new version of the OmpSs programming model is being developed, taking into account the nuances of newer technologies and architectures.

In this context, a need to develop support for heterogeneous devices for the new version of the model arises. This project presents the development and evaluation of support for CUDA enabled devices on the Nanos6 runtime being developed for the OmpSs-2 programming model. The design choices are analyzed in the context of modern GPU programming and architectures, and the performance and execution of a series of benchmarks is analyzed to understand the performance characteristics of the runtime.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context And Motivation

Heterogeneous computing has become prevalent as part of High Performance Computing in the last decade. One of the most common types of heterogeneous devices used are Graphical Processing Units. GPUs were initially developed for graphics processing, essentially computing thousands of pixels in a very short amount of time, however, with the rise in popularity of General Purpose GPU computing (GPGPU) manufacturers have begun creating frameworks and hardware specifically suited for HPC environments.

On the beginning of GPGPU graphical environments such as OpenGL Shaders were used to compute mathematical problems by mapping the problems to graphical parameters. Modern frameworks have made GPU computer user friendlier, but programability is still an issue, since GPU programming models differ from traditional CPU models. An average GPU program consists on moving the data from host memory to device memory, executing a GPU function (called *kernel*) and moving the data produced back to the host. The memory transfers must be done manually, which alongside other GPU specific nuances make programming on GPUs a relatively complex tasks.

Multiple programming frameworks have been developed for GPU programming, the most widely used being OpenCL[7] and CUDA[11]. A notable development on the latest versions of these models is the introduction of features concerned with programmability and ease of use instead of performance on this frameworks, for example, Unified memory in CUDA, a mechanism that provides automatic memory management, relieving the user from this task.

Frameworks focused in ease of use are not uncommon in High Performance Computing, were programability is considered one of the biggest issues to tackle in the coming years. Among such frameworks the OmpSs[4] programming model developed at the Barcelona Supercomputing Center can be found. This model provides programmers with a task based approach to parallel computing, where the user can execute an application by describing it as a series of tasks with dependencies among each other, leaving to the frameworks the responsibility of finding maximum parallelism.

In addition to regular CPU tasks, OmpSs supports a series of task models for different architectures, among them CUDA and OpenCL for GPUs. This allows a programmer to write a task based program capable of running on a GPU without

having to worry about any memory management.

OmpSs is based on two software components: the Mercurium compiler and the Nanos++ runtime. The compiler transforms C, C++ or FORTRAN code to an intermediate representation which uses the runtime to execute the tasks in parallel.

HPC devices, however, have become more specialized and advances and traditional programming models have to adapt to keep up. New features and architectures are introduced in the world of GPGPU and OmpSs needs to adapt in order to keep competitiveness. For this, a new version of the programming model is being developed, called OmpSs-2, as well as a new runtime to implement it, called Nanos6.

The reason for the development of a new runtime instead of improving on the existing system is because of several issues that can be found on the design and implementation of the current one. The main issue of Nanos++ is programability and maintainability; due to the development process followed in its development it is difficult and cumbersome to add new features and fix existing bugs on the code base. In addition to this, improving performance, flexibility and adding new features are the focus on the development of the runtime.

Due to this, instead of modifying the existing software, the new system is being built from the ground up, which creates a need for many of the features on the previous runtime to be redeveloped. Since an existing Nanos6 runtime is already functioning, the need to start re-implementing the different features of a runtime arise.

This project is concerned with the development of the support for CUDA tasks in the new runtime. Instead of just copying the functioning of the previous runtime, one of the goals is to re-imagine the existing design and to avoid its shortcomings as much as possible. In addition, since CUDA GPU architectures and language features have changed from the time in which the previous version was developed, the new features of CUDA will be used where adequate. For starters, this means that the new CUDA support is being developed using unified memory instead of regular memory management.

In addition to developing the feature a stated it is important to take into account that in the future more architectures need to be implemented Thus, it is desirable to create when possible an escalable and flexible system for asynchronous task execution, in order to ease the development of future projects. Besides this, a number of benchmarks and tests to evaluate the performance of the runtime must be developed, since no such suite exists for Nanos++.

The work on this project includes the development of support for the CUDA task execution on the Nanos6 runtime and the porting of several benchmarks to the OmpSs-2 model of programming, in order to evaluate the performance of the feature developed.

1.2 Document Organization

This document is divided into 6 chapters. Chapter 1 covers this introduction giving some context and motivation behind the project. In chapter 2 the relevant state of the art is presented, including GPGPU and Nvidia GPU architectures, the CUDA programming environment as well as the OmpSs programming model and the Nanos++ runtime.

Afterward, chapter 3 presents the methodology used to develop the project, together with the environment of the project, including a brief description of the existing Nanos6 runtime and a listing of the applications ported for this project. In chapter 4, Design & Development, the changes done to the runtime through this project are presented as well as a more in depth description of the components which have been modified.

Before the final conclusions and future work in chapter 6, the evaluation and results obtained in the project are presented in chapter 5, as well as the hardware environment in which the project is developed.

Chapter 2

State of the Art

Heterogeneous architectures have become popular in High Performance Computing on the latest years due to the performance they provide over CPUs on some applications. While the speedup that accelerators can provide is significant, programmability is an issue, since they are usually incompatible with traditional languages and paradigms, thus, a number of programming models focused on accelerators have been created.

In order to extract performance from accelerators it is vital to understand the nuances of the different architectures. In GPU computing in particular, being a rapidly changing field, is is particularly important to know the differences between the architectures of the different GPU devices and the features their programming models offer.

In this section, the relevant state of the art for this project is presented. Since the project is concerned with implementing GPU (specifically CUDA) support on the Nanos6 runtime, the latest Nvidia GPU architecture, the latest version of the CUDA runtime and the previous version of the OmpSs runtime are presented.

2.1 GPU Architecture

The capabilities of Graphical Processing Units for general purpose computing stem from their architectural difference over traditional CPUs. GPUs were originally designed solely for graphics computing. This means that the only task they were created to performs is pixel processing, comprised of completely parallel huge number of small operations.

The traditional architecture of CPUs is composed of a few powerful cores with a lot of cache memory that can run a few software threads at the time. In contrast, GPUs are composed by hundreds or thousands of small cores, with little to no cache memory, capable of running a big amount of software threads at the same time. Broadly speaking, GPUs are optimized for taking huge batches of data and performing the same operation over all of the data simultaneously.

CUDA GPU architectures are composed of several Streaming Multiprocessors (SM) which run the GPU functions, called *kernels*. Each of the SMs is composed of several CUDA cores and receives a single instruction, thus, each of the processors inside a SM runs the same instruction over multiple data, forming a Single Instruction Multiple Thread (SIMT) execution model. A scheme of the architecture of a GP100, the first Pascal CUDA GPU can be seen on figure 2.1. In this figure it is

possible to appreciate the distribution of several SM over the chip as well as a small cache, a feature included to newer GPU architectures.

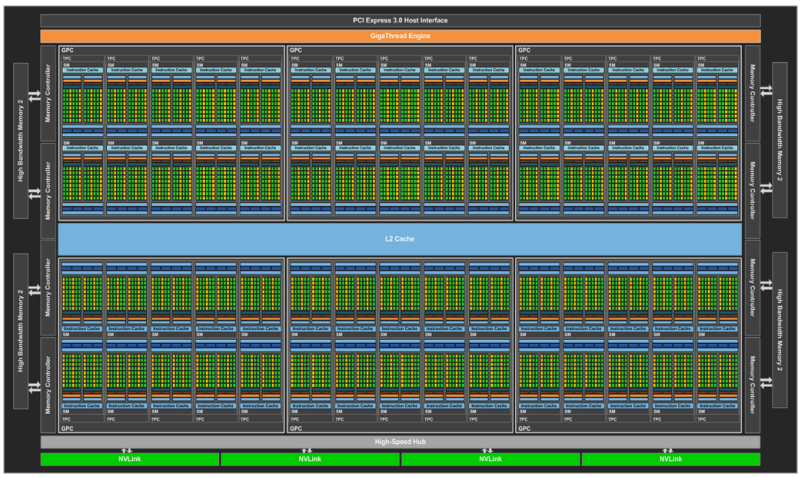

Figure 2.1: The architecture of a Pascal GP100

2.1.1 CUDA Pascal Architecture

Pascal is the latest micro-architecture developed by NVIDIA, as of 2017. Released in April 2016 it is the successor of the previous Maxwell architecture. The SMs of this architecture contain each 64 CUDA cores, partitioned into 2 blocks, each having 32 single-precision CUDA Cores, an instruction buffer, a warp scheduler, 2 texture mapping units and 2 dispatch units. An scheme of a Pascal SM can be found on figure 2.2.

Figure 2.2: The architecture of the Streaming Multiprocessors of a Pascal GP100

In addition to the architecture changes, Pascal comes with a new version of CUDA, CUDA 8. This new version of the runtime includes numerous changes to the Unified Memory system. The most significant of these changes is concerned with

how the data needed to run kernels is transferred. In previous versions of CUDA, when a kernel with managed memory accesses was run, all managed data, even the one not needed by the kernel, has to be copied before the kernel can be run, which can be a major issue for performance. In Pascal *GPU Page Faults* are implemented, allowing kernels to launch without any transference, instead moving data when it is needed on the GPU, though this comes with a penalty for performance, since the CPU must process the page faults.

Besides this, there are other improvements on the Unified Memory. Since Pascal, it is possible to run code on the CPU using managed data while a GPU kernel is running (though it can cause severe performance issues), and several optimization functions are provided such as a prefetch function (**cudaMemPrefetchAsync**). All in all, Pascal makes it possible to optimize programs using unified memory, allowing it to move beyond just a prototyping framework.

2.2 CUDA

Compute Unified Device Architecture (CUDA)[11] is a parallel computing platform and programming model developed by Nvidia for general purpose computing on CUDA-enabled GPUs. This means that the whole CUDA toolkit, including a compiler, computing libraries, analysis tools, etc. is limited to Nvidia cards, which is generally regarded as the main disadvantage of the model.

To use this model, the programmer has to write specialized functions (called *kernels*) which are executed concurrently on the Graphics processor. In order to run a kernel, the programmer calls a function annotated with a *__global__* keyword through a specialized syntax which indicates the number of threads needs to run in the GPU (in the forms of threads and blocks).

The programmer is not only responsible for writing the computational kernel, but also of performing memory allocation and transfer between host and device memory, though this can be leveraged from the programmer with a mechanism introduced in the 6^{th} version of CUDA called Unified Memory. A simple CUDA application generally involves the following steps:

1. Initializing the data on the CPU
2. Copying the data to the GPU
3. Launching the kernel
4. Copying the generated data back to the host
5. Checking/Storing/etc. the output data

In general a CUDA programmer needs to deal with some key factors for performance. One of the most important is minimizing the memory movements in the application. The cause for this is that generally the slowest portion of the architecture is the connection between the Host and the CPU, thus reducing the amoung of data movement is crucial. The programmer also needs to deal with other mechanism such as the usage of shared memory in order to reduce the number of (expensive) accesses to global memory.

```
// Declaration of kernel
__global__ void saxpy(int n, float a, float *x, float *y);

// Memory allocation: Host
x = (float*)malloc(N*sizeof(float));
y = (float*)malloc(N*sizeof(float));

// Memory allocation: Device
cudaMalloc(&d_x, N*sizeof(float));
cudaMalloc(&d_y, N*sizeof(float));

// Memory transfer
cudaMemcpy(d_x, x, N*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N*sizeof(float), cudaMemcpyHostToDevice);

//Kernel launch
saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d_x, d_y);
```

Listing 1: CUDA pseudocode

Additionally, the CUDA toolkit provides a series of computing libraries such as cuBLAS (Basic Linear Algebra Subroutines)[10] and cuFFT (Fast Fourier Transformations)[12] as well as a series of functions to run programs on multiple CUDA-enabled GPUs.

2.2.1 CUDA Streams

CUDA streams are a concept central to the execution of CUDA operations. By default all CUDA functions are synchronous, except for kernel executions which need some form of synchronization. It is possible to execute different operations asynchronously and concurrently by using a mechanism called CUDA stream.

A stream acts as a queue to which operations are submitted, including asynchronous variants of regular CUDA function calls as well as kernel functions. Tasks sent to a queue are executed sequentially, however, they can be run in parallel with any operation in a different stream. This allows to potentially occupy all the SMs of the GPU device as well as using the different copy engines simultaneously. In addition, the FIFO queue like behaviour of streams allow for implicit synchronization of operations if used correctly, for example, by issuing to the same stream a transfer of data for a kernel and the kernel in order.

2.2.2 Unified Memory

Unified Memory is a mechanism introduced with the release of CUDA 6 which allows the programmer to avoid the task of memory management. Traditionally, the CPU memory and the GPU memory are separated in the programming model and the

program explicitly copies memory from one to another, calling functions such as *cudaMemcpy*.

Unified Memory creates a pool of managed memory shared between host and device. This memory can be accessed by all devices with a single pointer, in contrast to regular memory which needs a host pointer and a device pointer (see figure 2.3. In order for this to work, the system automatically migrates data allocated in this pool of managed between the devices.

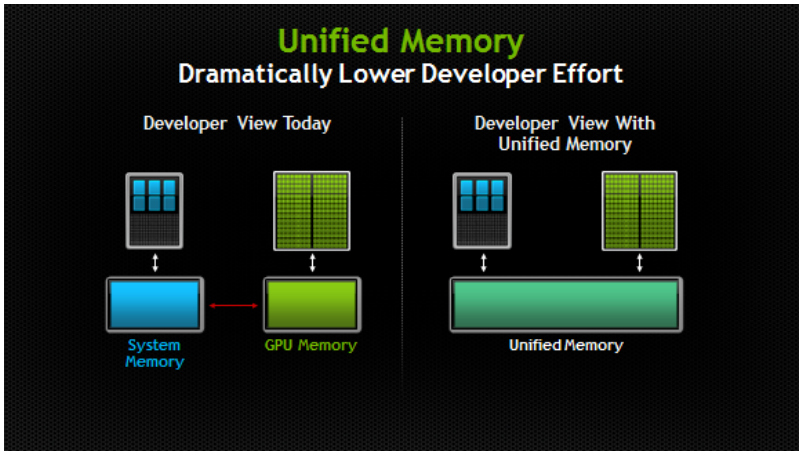

Figure 2.3: Unified memory as conceptualized by Nvidia

Using Unified Memory means that the programmer sacrifices performance for productivity due to which it is often used as a prototyping tool. With more recent versions of CUDA, emphasis has been made in increasing performance and tuning primitives have been included in unified memory.

In the newest version on the CUDA toolkit (version 8) intorduced with the Pascal Architecture, Unified Memory has been improved with the usage of GPU page faults, enabled by the new architecture. GPU page faults are used to trigger transfers,instead of batch copying of the whole unified memory pool on kernel execution). Together with this new optimization and tuning functions, for example, *cudaMemPrefetchAsync*, have been added. In addition, a support mechanism for certain operating systems is in development in order to enable allocation of managed memory with the default OS allocator (i.e. malloc).

With this changes it seems that there is a shift in focus on the Unified Memory from a tool for prototyping to a mechanism that can be used for the development of well performing and production ready programs.

2.3 OMPSS Programming Model

The OmpSs programming model [4] is an effort to extend OpenMP [1] with new directives to support asynchronous task-based programming and heterogeneity.

OmpSs takes ideas from StarSs [6], and enhances OpenMP with support for irregular and asynchronous parallelism as well as heterogeneous architectures, while including StarSs dependence support and data-flow concepts allowing the runtime to automatically move data between devices as necessary and perform different optimizations.

OmpSs aims to ease the development of applications for HPC systems while keeping a good performance. For this, the framework uses compiler directives (similar to OpenMP) to annotate the code with tasks and dependencies. This approach allows to produce a parallel version of programs without affecting the semantics of the applications.

Execution Model

OmpSs uses a thread-pool execution model. There is a master thread, which starts the execution of the application and several worker threads which cooperate in order to execute the work it creates through **Task** constructs. The parallelism of applications is obtained from these **task** constructs at runtime, and their data dependencies.

During execution, a data dependency graph is generated from the information extracted from dependency clauses and their directionality (`in`, `out` and `inout` clauses). The dependency mechanism is the key for the correct running of tasks, allowing that with no dependencies or only read dependencies to run concurrently.

Memory Model

From the programmers point of view, there is a single address space, in order to simplify programming. Internally, however, they may be multiple address spaces, due to for example, the presence of heterogeneous devices such as GPU. Any data that has been marked with the OmpSs syntax can be safely accessed by the parallel regions of the program (tasks) though safety is only guaranteed for the sequential code after a synchronization point. The runtime takes care of the management of data, keeping track of the location of any region of memory and transferring data to the location it is needed in.

Syntax

OmpSs annotates code by marking regions of code, such as functions, with C compiler directives. These directives are formed with the `#pragma omp` or `#pragma oss` clauses followed by the specific syntax of the directive.

Tasks

Tasks are specified with `#pragma omp task` followed by a specification of the dependencies in a region of code, for example, at the declaration of a function. The dependencies are specified with 3 clauses, `in` for input dependencies, `out` for output dependencies and `inout` for input and output dependencies. These clauses allow specifying arrays and pointers, the data addresses and sizes do not need to be constant since they are computed at execution time.

Taskwait

The taskwait directive, constructed with `#pragma omp taskwait` is used as a synchronization point. When the runtime encounters this clause, it will wait for all the pending tasks to finish executing before continuing the execution. In addition all

```
#pragma omp task in([n]x) inout([n]y)
void saxpy(int n, float a, float* x, float* y);
```

Listing 2: OmpSs task construct

data that has been moved around is flushed to their origin addresses, since the semantics of the directive specify that all data can be accessed by the sequential code after a synchronization point. However, taskwait can be provided with a `noflush` clause to avoid data transfers.

Target construct

The target construct is used to support heterogeneity. The directive is applied to tasks (in a separate pragma) and specifies the type of device as well as what data needs to be copied. The construct is formed as `#pragma omp target device(...)` and can be followed by `copy_in`, `copy_out` or `copy_inout` to specify which data has to be copied to and from the device. These can be replaced by `copy_deps` to indicate that all data specified in the dependencies must be transferred. CUDA and OpenCL tasks must also indicate the parameters for the execution (Gird and Block dimensions for CUDA) with the `ndrange` clause.

```
#pragma omp target device(cuda) copy_deps ndrange(1, n, 128)
#pragma omp task in([n]x) inout([n]y)
void saxpy(int n, float a, float* x, float* y);
```

Listing 3: OmpSs task construct

Components

OmpSs relies on two components, the Mercurium source-to-source compiler [15] and the Nanos++ parallel runtime [16].

Nanos++

Nanos++ is a runtime library that supports the OmpSs programming model. It is responsible for the scheduling and execution of tasks following the constraints specified by the user, such as dependencies. Nanos++ supports a number of different architectures and scheduling mechanisms. The runtime creates the different threads, calculates and resolves the dependencies, manages the data transfers and runs the tasks.

Architectures

Currently OmpSs supports several different architectures:

smp: General purpose CPUs.

smp-numa: Non-Unified Memory Access (NUMA) systems with general purpose CPUs.

gpu: CUDA-capable GPUs.

opencl: Any architecture supporting OpenCL (GPUs, CPUs and Xeon Phi).

fpga: FPGA devices.

hstreams: Any architecture supported by the hStreams library (Xeon Phi).

cluster: A cluster system with several nodes, can be combined with any of the previous architectures.

mpi: A cluster system, using the Message Passing Interface (MPI) to distribute tasks among nodes.

Task flow

When the runtime encounters a piece of code annotated as a task, it creates a `task` structure inside and captures its environment, data and dependencies. For the execution, a tasks life can be divided into 5 stages:

Instantiated: When the data is created, the runtime computes the dependencies and adds a node to the dependency graph representing this task.

Ready: A task is ready when all its data dependencies are satisfied and it is ready to be executed on a available resource. At this stage is when task scheduling occurs.

Active: A task is active when all the scheduling operations for a task have been done but before the task is run. At this stage, the coherence mechanisms of the runtime are accessed to ensure that all necessary data is available and to perform transfers.

Run: The task is being run.

Completed: When the task finishes running, output data is processes by the coherence layer and the dependency system updates the graph, marking all the tasks whose dependencies have been resolved as ready.

Mercurium

Mercurium is a source to source compiler, aimed at fast prototyping. Its main purpose is to be used as a compiler for OmpSs and it supports C, C++ and Fortran.

The role of the compiler is to recognize the directives of the language and transform them into Nanos++ runtime calls. The compiler is also responsible to restructure code for different target devices, and of invoking the native compilers of each of the target architectures. The compiler is able to generate several files for different device-specific code, however all the binaries and files are merged into a single object file, recovering all information on the linkage step, in order to maintain the traditional compiler behaviour of generating a single executable.

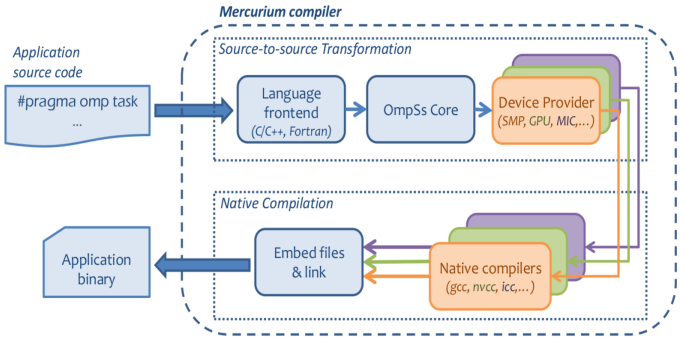

Figure 2.4: Mercurium compilation flow

GPU Support

The Nanos++ GPU architecture is based on a event driven flow [9] based on the scheme in figure 2.5.

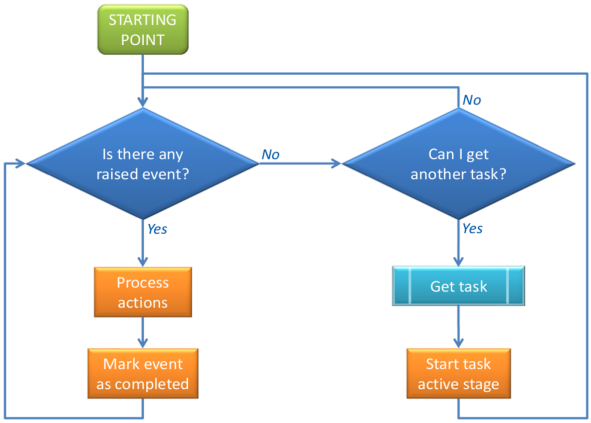

Figure 2.5: Nanos++ GPU event driven flow

In order to implement this scheme, Nanos++ creates separate helper threads for the heterogeneous architectures, dedicated to launch tasks and copy operations, as well as checking for the synchronization events and available tasks. When a task or a memory operation is launched, an event is created on the system and the helper thread will poll it until it has finished in order to launch the following operation.

The CUDA support system uses multiple streams for execution. One stream is dedicated to input transfers and another one to the output transfers, separate from a pool of streams for kernels with a variable number configurable by environment variables. This separation of streams makes the usage of events and polling on the CPU a necessity. More details on this are given in section 4.2.4.

In order to operate, the runtime keeps track of the location of all the data. CPU regions of data are transferred via regular CUDA copy operations to a pool of GPU memory that is pre-allocated at start-up.

The Mercurium compiler creates a wrapper function when a CUDA kernel is called. Inside this wrapper, the launch parameters of the kernel are calculated (grid size and block size), a runtime API function is called to obtain a execution kernel and the kernel is launched. This wrapper functions are implemented on a intermediate

file dedicated to CUDA code, which is compiled with the Nvidia CUDA compiler before being merged with the rest of the program.

Chapter 3

Environment and Methodology

3.1 Methodology

The project has been carried out through an iterative prototype based development process. After a initial design phase, the works has followed a structure of developing a minimal working system, evaluation and iterating on the design.

The initial design of the runtime has been done with the feedback of developers of Nanos6 and Nanos++, trying to avoid the mistakes the previous runtime did and considering the nuances of the new runtime being developed.

The progress of the project has been tracked via weekly meeting when possible, where the progress and results obtained as well as the tasks to do were discussed.

Design phases consist on analyzing the results of previous evaluations and testing some considered feature, as well as discussing how a feature can be implemented in the weekly meetings. Implementation phases mostly consisted on writing C++ code on a branch of the Nanos6 project hosted by the BSC, as well as occasionally talking with the main developer of the system in order to understand the inner workings of components crucial for the implementation of CUDA support. Evaluation phases, on the other hand, involve porting application to OmpSs and OmpSs-2 as well as running the applications on a machine and analyzing the results.

A iterative approach was chosen because the project is part of a bigger project with many developers involved and it requires many features to be implemented, many of which are still works in progress. By using a more agile iterative development process it is possible to adapt to changes and new developments performed by other developers (such as a hierarchical scheduler developed in parallel to this project) and to build a working system without the need for it being feature complete, as well as quickly adapting to issues found with the design chosen.

Application Porting

A number of applications have been ported to OmpSs-2 for the evaluation of this project. The porting of the non OmpSs applications has followed the following steps.

1. Determine the relevant sections to convert into tasks.
2. Separate the kernels into a separate `.cu` file from the rest of the code
3. Create a header file for the kernel in which the task is annotated

4. Remove the CUDA memory transfers on the code and substitute the kernel launch for a direct function call to the function in the header mentioned

On the other hand, already existing OmpSs+CUDA applications only require modifications to their pragmas to update them to the current version of the runtime, as well as working around the features that are not yet implemented, though no such case has been found.

Besides this, when executing applications before the compiler support for CUDA tasks is implemented, additional steps are needed to run. In order to work around this, the tasks are masked to the compiler as SMP task and the intermediate files are kept using the `-K` flag on mercurium. In the intermediate files, the objects in which the task information of each task is kept are modified for the CUDA tasks are modified with a task type indicator, and the declaration of functions called by the runtime is modified with any parameter introduced only on the runtime. Trough this process, it is possible to run OmpSs v2 CUDA applications without compiler support.

3.2 Environment

3.2.1 Nanos6

Nanos6 is a runtime created to run underneath a new specification of the OmpSs programming model, called OmpSs version 2. Nanos6 is the next version of the existing Nanox (also called Nanos++) runtime, and the goal of its development is to remake the runtime from the ground up to avoid the mistakes that may cause performance issues in the previous version as well as designing the runtime in a way that is more expandable and maintainable than it was. This project is part of the development project of Nanos6.

As of now, a first functioning version of Nanos6 exists and offers some basic functionality, though it is lacking some of the functions available in Nanox, such as heterogeneous device support. The first asynchronous task type to be added are CUDA task, and the work to develop this functionality is done through this project.

The system can be divided into the following components, considering how the code base is currently divided:

API: The external interface of Nanos6, which contains the functions called by the code generated by mercurium in order to create task, register tasks, synchronize, etc. There is additionally a debug API for developers.

Loader: The loader is a component used to load the version of the runtime to use. This is needed because the compilation process of Nanos6 generates different binaries according to the version, compiled with different flags.

Dependency Manager: The dependency manager is the component is charge of tracking the dependencies of the tasks and unblocking those whose dependencies are solved.

Executors: The executors are the elements of running the code of the tasks, be they SMP task, CUDA kernels or any supported architectures. This component is the one "running" the application, including the worker threads, etc.

Scheduler: The scheduler determines the policy used to select which tasks are run first in the limited computing resources in the system.

Hardware Information: The hardware info is in charge of reading the information of the underlying hardware of the system in order to provide a model that the runtime can use to optimize the execution.

Instrumentation: The instrumentation component implements the functions needed to instrument the execution of a OmpSs 2 application on environments such as Extrae[8].

Tasks: `Task` objects are used to represent the tasks that the user annotates on the application. The data structures contain the data necessary for its execution, such as the user function and the dependency data, as well as internal variables to track and log its status through the execution of the program.

System: This portion of the code contains the implementation of the functions on the external API as well as the start up and shut down portions of the system.

Low-level: The low-level components contain different implementations of necessary elements for the functioning of the runtime, such as spin-locks.

A more in depth description of some sections of the runtime; as well as some nuances of the OmpSs second version for which the runtime has been developed follow. Chapter 4 delved into the changes and new functionality implemented as part of this project as well as some more in depth explanations needed to understand the implementation. The goal of the descriptions on this section is to give a broad idea of parts of the system that are not affected by the work on this project.

Thread Model

Both Nanos6 and Nanos++ use a pool of threads for the execution, creating (by default) as many threads as processor cores in the system. Threads will run iteratively executing the available tasks obtained from the scheduler, however, similarities end there. In Nanos++ tasks can be paused and can be resumed in any of the available threads, while in Nanos6 tasks are assigned a thread and will be resumed on the same threads. Additionally, Nanos6 threads will become asleep when no tasks are available for execution (such as when there are less tasks that threads on the system) and will be resumed when new tasks become ready.

Regarding asynchronous executions, Nanox uses a number of *Helper Threads* in order to track the different events created by the asynchronous tasks and to keep the coherence of the application. A thread is created for each device of a non SMP architecture in use and it runs a number of helper tasks (to launch asynchronous tasks, check for task completion, etc.) iteratively. The problem with this approach is that the helper thread will compete with the SMP worker threads, and can limit the performance obtained. In order to tackle this, the SMP threads on Nanos6 will also be in charge of running the tasks of the asynchronous helper threads on the previous runtime, thus eliminating the need for helper threads. In order to do this, the helper tasks are run when the worker threads are idle or in between task executions.

Task Execution Model

The task model in Nanos6 is comprised of a number of states through which a task in Nanos6 goes during its lifetime. Currently the following states exist:

Blocked A task is blocked (by others) when its dependencies are not yet solved. This is the starting state of tasks when they have dependencies, however, it is possible that a running task becomes idle when a taskwait is found in its code, since tasks can spawn other tasks. The dependency manager component of the runtime is responsible for tracking blocked tasks.

Ready When its dependencies are solved or the taskwait is satisfied a task is becomes ready and it is sent to the scheduler. The scheduler is responsible of choosing which of its ready tasks will be run first.

Running A task is running once it starts executing its *body* (the user level code). Once it is finished running, the responsibility of notifying the dependency system once it is finished corresponds to the computing resource it is running on.

Finished Once a task has finalized its execution, it is considered finished.

The diagram on figure 3.1 illustrates the task state flow.

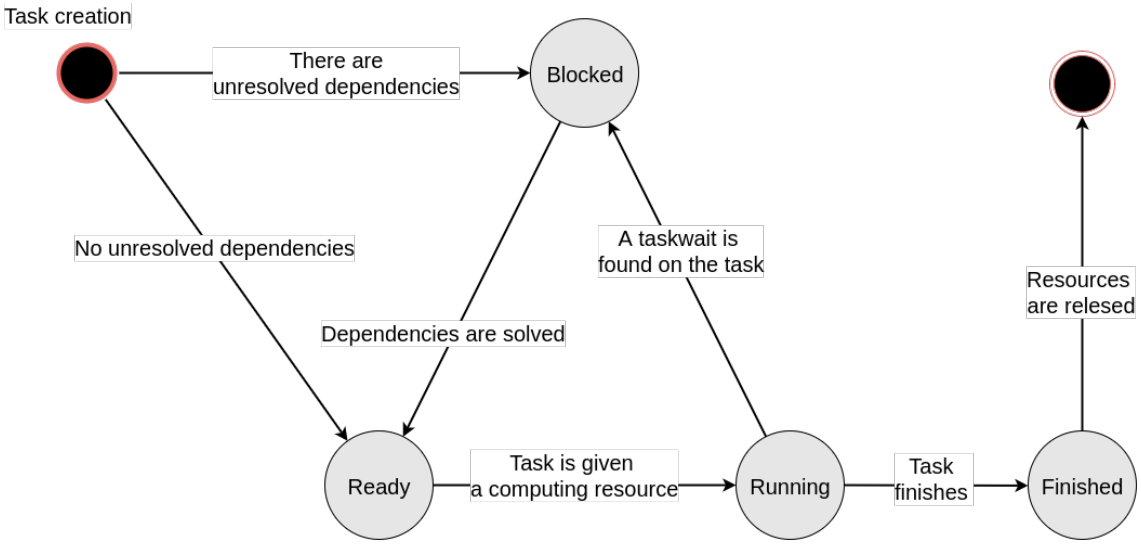

Figure 3.1: Task states of Nanos6

Dependency Manager

The dependency manager is the component of Nanos6 which takes charge of solving the dependencies of the system. It exposes some functions to be called by the threads of the runtime when a task is solved in order to update the pending tasks and unblock them if necessary. It is also accessed by the API functions used to submit tasks, in order to either add it to the list of blocked tasks or to be sent to the scheduler directly.

There are 2 different dependency models at the moment:

- Discrete regions
- Linear regions

The dependency model using discrete variable is only capable of identifying dependencies by the variable name (base pointer) without considering subsets of the data. Due to this limitation, most of the work is currently done on the Linear Regions model, which allows to specify dependencies by using a base pointer as well as size or array subset indications.

Scheduler

The scheduler of the runtime determines the policy used to determine the order in which ready tasks (those whose dependencies are solved)

Several schedulers are available for Nanos6, which are selected using a environment variable called `NANOS6_SCHEDULER`. There are several different schedulers, among them:

- A Naive Scheduler which executes tasks in a LIFO manner without taking into account neither locality or priority.
- A Immediate Successor Scheduler which gives priority to the successor of a task to be run the CPU that ran the antecessor in order to exploit locality.
- A simple FIFO Scheduler.

In addition to the regular schedulers, a number of polling schedulers are also available which re-implement the policies of the available schedulers, providing a different querying mechanism to the rest of the runtime. For regular schedulers, the threads will query to see whether a task is available for them, and if there is none they will become idle. If a polling scheduler is used and a task is not available for a thread, that thread will be given a *slot* data type which will be updated with a task when one is available, allowing the thread to poll instead of becoming asleep.

Additionally, a hierarchical scheduler has been developed in order to provide more flexible development and adaptability over the monolithic development used up to now. In order to do this, a `HostHierarchicalScheduler` type of scheduler is added, as well as `NUMAHierarchicalScheduler` in order to build a tree of schedulers.

The design aims to have a *host scheduler* with contains a number of schedulers per each NUMA node on the system. The scheduler for each NUMA node can be configured differently, regardless, the tasks are first redirected to the most appropriate node according to the NUMA scheduler policy, where it is sent to a concrete CPU depending on the policy of the scheduler on that node. A possible tree for a system is illustrated in figure 3.2.

OmpSs 2 Changes

Some changes have been made to OmpSs version 2 programming. Most of them come from the lack of features present in the previous version due to them being yet unimplemented. Some changes are planned for the future, for example, a clause equivalent to the current ***`ndrange`*** of CUDA/OpenCL tasks which allows to use a more CUDA-like notation. In addition the default pragma name has been changed from `omp` to *oss*, which will be reflected in all the following code listings.

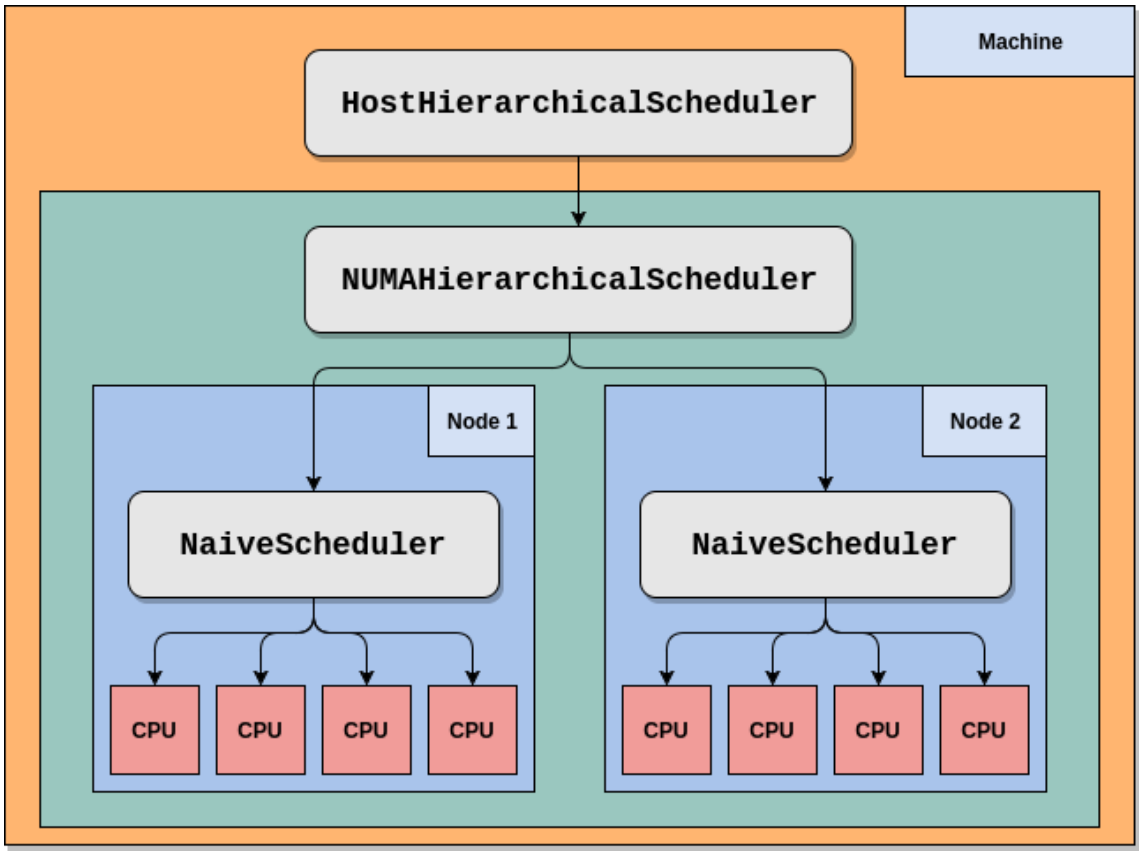

Figure 3.2: A hierarchical scheduler tree for a 2 NUMA Node machine with no asynchronous devices

3.2.2 Applications

For the testing and evaluation of the work developed a series of applications have been ported to OmpSs and OmpSs-2. The applications can be categorized into 2 groups, applications previously existing on the BSC application repository[14] and OmpSs learning examples[17] and benchmarks ported from the Rodinia benchmark suite[3].

In addition a microbenchmark has been developed to test the behaviour of the GPU on certain scenarios related to the functioning of Nanos6.

This section provides a brief description of the different benchmarks used for the application, along with a depiction of the task graph in some cases. Graphs are not shown for all applications since the outline of some applications is similar or identical to others, so only those with significant differences are shown.

BSC Application Repository and OmpSs examples

The applications on the BAR are written using the OmpSs model. Some modifications have been needed due to the applications having been developed some years ago and the model having changed slightly. Besides that, the only additional changes needed for testing involve modifying the directives to OmpSs version 2, introduced with nanos6.

Saxpy SAXPY stands for *Single-Precision A X Plus Y*, which is a combination of scalar multiplication and vector addition. It is a very simple operation, it takes two 32-bit float vectors (X and Y) and a scalar value (A). It multiplies each element of X by A and adds it to Y. The kernel used is the following:

```
1  __global__ void saxpy(long int n, float a, float* x, float* y)
2  {
3     long int i = blockIdx.x * blockDim.x + threadIdx.x;
4     if(i < n) y[i] = a * x[i] + y[i];
5  }
```

Listing 4: CUDA Saxpy code

Since there are no dependencies between the vector elements, in OmpSs, the vector is divided into equal sized chunks and the execution of each chunk becomes a task. There being no dependencies, it is a good application to evaluate the ability of the runtime to handle many concurrent tasks and compare it to the one on the previous runtime. The dependency graph of the application is quite simple, as is shown in figure 3.3.

Multi-Saxpy Multi-Saxpy is a modified version of the previous application developed specifically to test a specific scenario for Nanos6. Its is to test the memory transfer capability, thus the application describes the worst case for memory transfers. This micro-benchmark runs a number of SAXPY kernels iteratively, and between each iteration it modifies the data on the CPU.

The goal of the application is to create a situation where data needs to constantly move back and forth from the GPU to the CPU. Each run of the kernel and CPU modification is divided in blocks of equal size, a task being created for each block. The application serves no particular purpose in the mean that it does not produce any useful results, however, it is useful to test the performance of the runtime. Its task graph is shown in figure 3.4.

Nbody Simulation The N-Body simulation is a molecular dynamics computation where a system of bodies (atoms and molecules) interacts for a period of time. The result of the simulation gives a view of the motion of the bodies whose trajectories are determined by forces between bodies and their potential energy.

The data dependency graph for this application is shown on figures 3.5.

Matrix Multiplication This application performs a dense matrix multiplication of two matrices and stores the result into another matrix: A * B = C. For simplicity, all matrices are square matrices and are divided into square tiles as well. Matrix multiply is a well-known algorithm that requires a significant amount of data movements.

Perlin Noise Perlin Noise is a algorithm used to increase the realism on computer generated images. The application involves 3 steps which are computed for each pixel

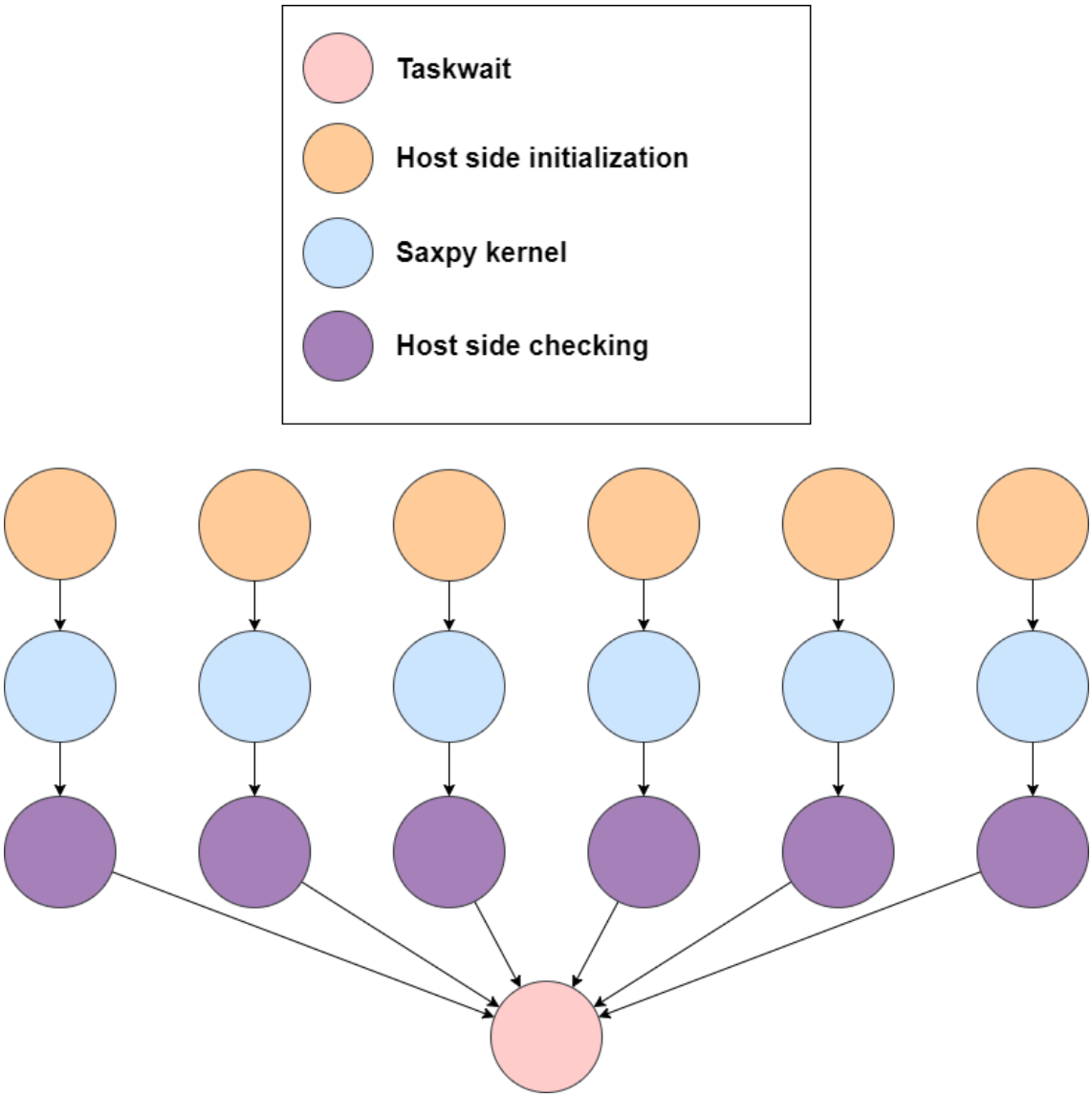

Figure 3.3: Saxpy task graph

of the image; grid definition, computation of the dos product between the distance gradient vectors and interpolation between the values.

In the OmpSs version, a input image represented as a two dimensional array is divided into several stripes containing multiple rows of pixels, with each task computing the gradient noise for one of these chunks.

Krist This application is used on crystallography to find the exact shape of a molecule using Rntgen diffraction on single crystals or powders. It computes crystallographic normalized structure factors. For this, data is kept on 3 arrays, two of which are read intro the GPU while the other is used to store the results.

In the OmpSs version, the arrays are divided into chunks and the computation is performed independently, with almost no data sharing between the tasks.

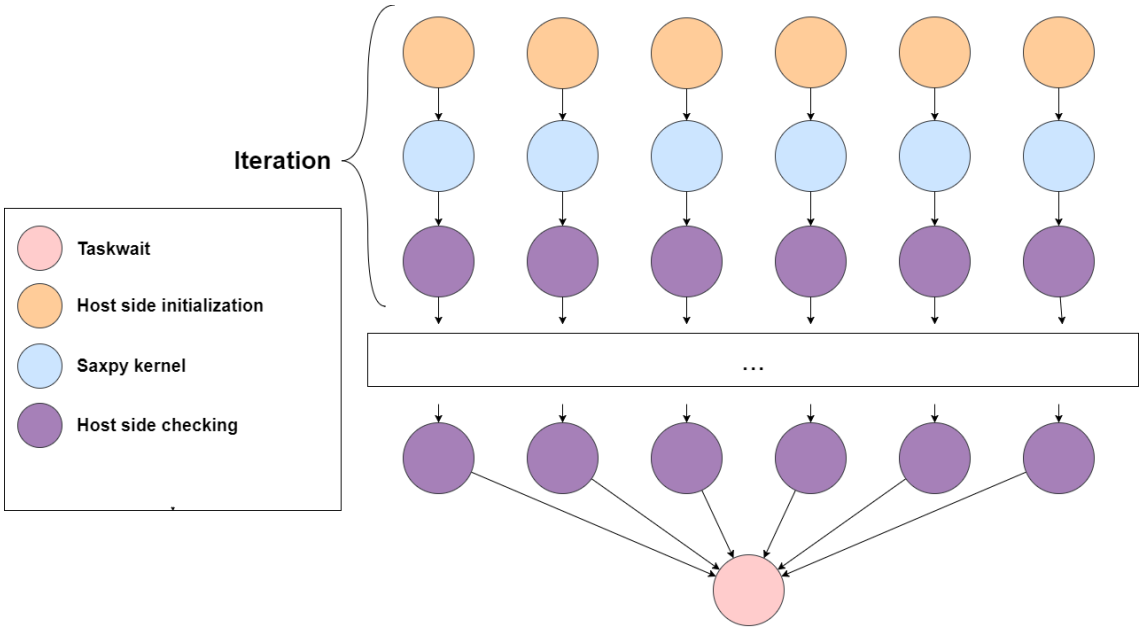

Figure 3.4: Multi-Saxpy task graph

Rodinia Benchmarks

Rodinia[3] is a benchmark suit focused on heterogeneous computing, created to tackle the lack of standardized heterogeneous computing benchmark suites.

Its applications are written in CUDA (as well as OpenCL), thus they need to be ported to OmpSs. This requires identifying the GPU related regions as well as the potentially parallel CPU regions and creating pragmas to parallelize them. In addition, any transfer operation must be removed, since these operations become responsibility of the runtime, and kernel launched must be changed to direct (annotated) function calls.

Only 8 of the Rodinia CUDA benchmarks have been translated and evaluated due to time constraints.

Breadth-First Search The BFS benchmark provides a GPU implementation of this graph traversing algorithm. The application is formed by two kernels which are run iteratively one after the other, the first used to traverse the graph and the second to determine a stopping point for the algorithm..

Gaussian Elimination Gaussian Elimination computes result of a liner system row by row, solving for all of the variables. The algorithm must synchronize between iterations, however the computation in each iteration can be processed in parallel..

Hot-Spot Hot-Spot is a tool used to estimate processor temperature based on the architectural design and simulated power measurement. A single kernel is used to calculate the temperature iteratively, however, on the OmpSs version, data is divided in blocks an a kernel is launched for each block. The process is repeated iteratively, using the dependencies of the tasks to synchronize without the need for a explicit synchronization..

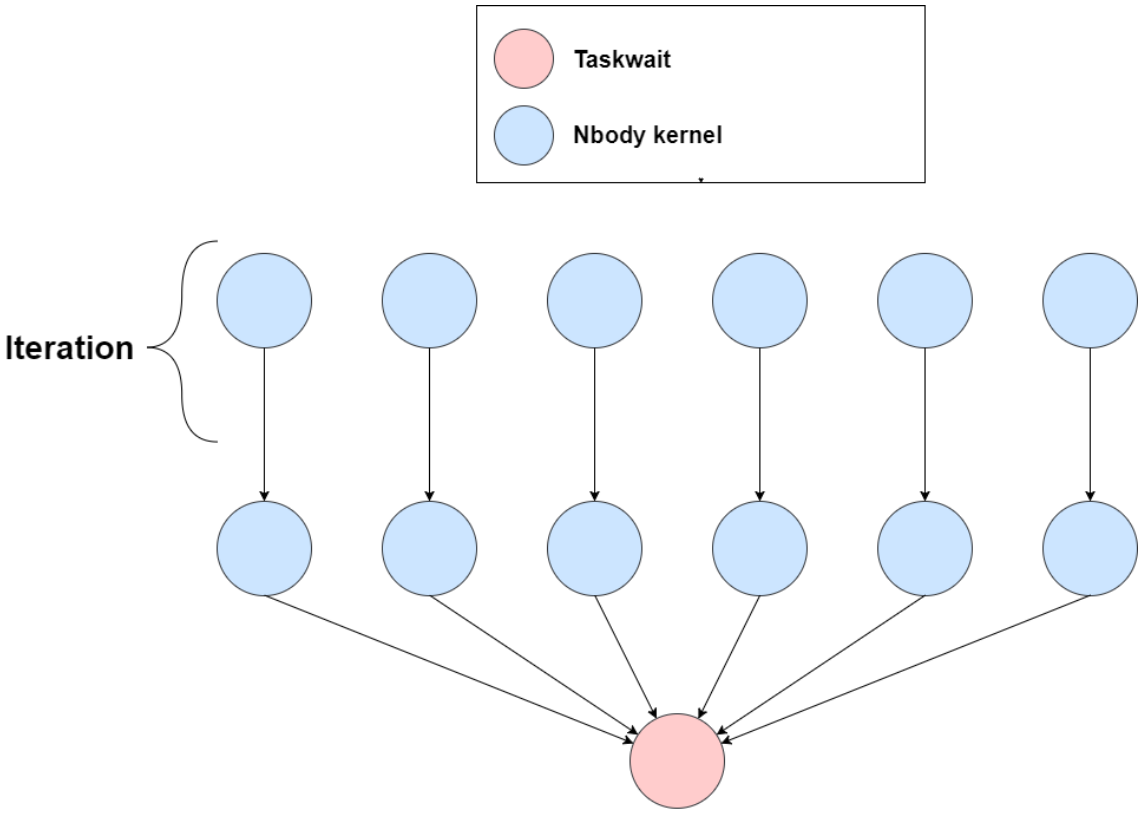

Figure 3.5: Nbody task graph

k-Nearest Neighbours The algorithm finds the k-nearest neighbors from an unstructured data set. The sequential version algorithm reads in one record at a time, calculates the Euclidean distance from a target latitude and longitude, and evaluates the k nearest neighbors. The CUDA version reads many records at the same time and calculated the euclidean distances on multiple kernels (records are calculated in separate blocks) and the CPU updates the list of nearest neighbours. The task graph is shown on figure 3.6.

Needleman-Wunsch NW is a nonlinear optimization method for DNA sequence algorithms. The potential pair of sequences are organized into a 2D matrix. First, the algorithm fill the matrix top left to bottom right, where the optimum alignment is the path through the array with maximum score, the score being the value of the maximum weighted path ending at the cell. In the second step, the maximum path is traced backwards to deduce the optimal alignment.

The computation is done by running two kernels, one for each step. Both kernels are run in separate iterations, first the one corresponding to the filling step and then running the one corresponding to the back-trace step.

Particle Filter Particle Filer[5] is a statistical estimator of the location of a object given noisy measurement of the location and an idea of the object's path in a Bayesian framework. The PF begins tracking a object by making a series of guesses given what is already known, The likelihood of those guesses is determined using a predefined likelihood model. Then the guesses are normalized based on their likelihoods and the normalized guesses are summed to determined the object current

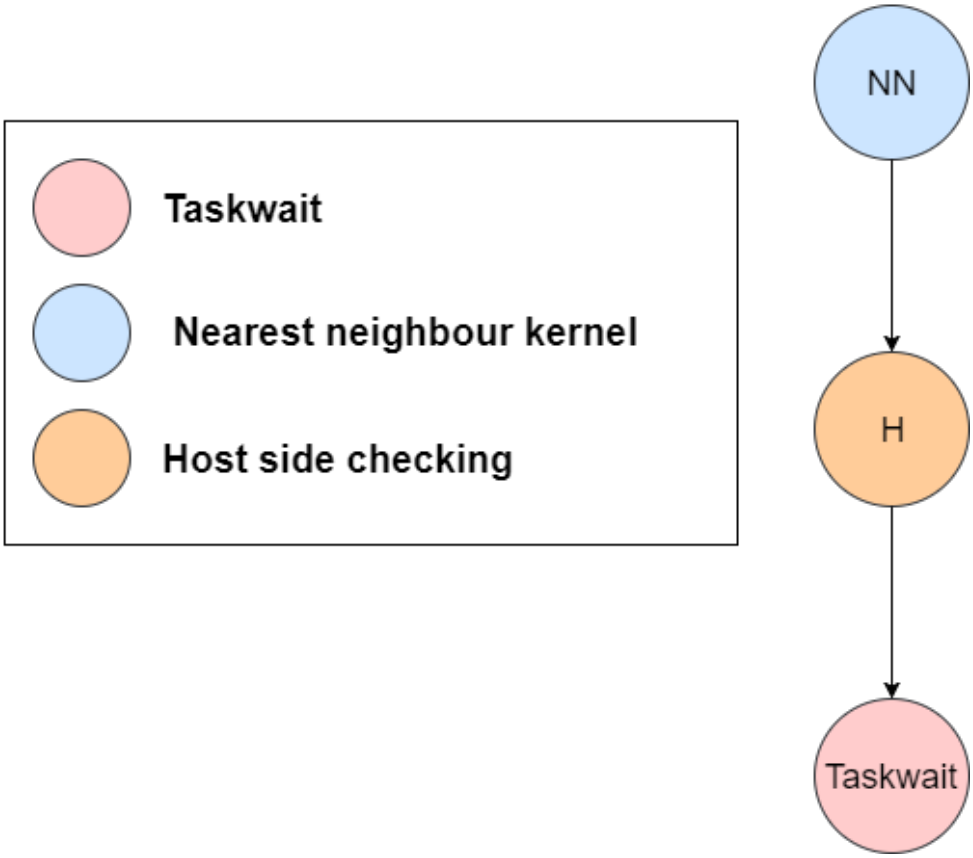

Figure 3.6: k-Nearest Neighbours task graph

location. Before repeating the process for all the remaining frames in the video, the guesses are updated based on the determined location.

Four kernels are created for the CUDA version, one for each mentioned operation; likelihood calculation, guess normalization, sum of normalized guesses and guess updating. The dependency graph of the application is illustrated in figure 3.7.

CUDA Microbenchmark

A GPU microbenchmark was developed for this project in order to evaluate different approached to GPU execution. This was initially done to find a problem causing performance losses when performing some optimizations in CUDA unified memory; this is described in chapter 5.

The microbenchmark executes a modified version of the Saxpy code. The most notorious modification is that it is not executed in OmpSs, but rather it mirrors the ”divide in blocks to create various tasks” approach used in OmpSs. In addition, the execution is done iteratively, the goal not being performing a specific operation but rather testing a worst case scenario for Nanos in which a program requires all data

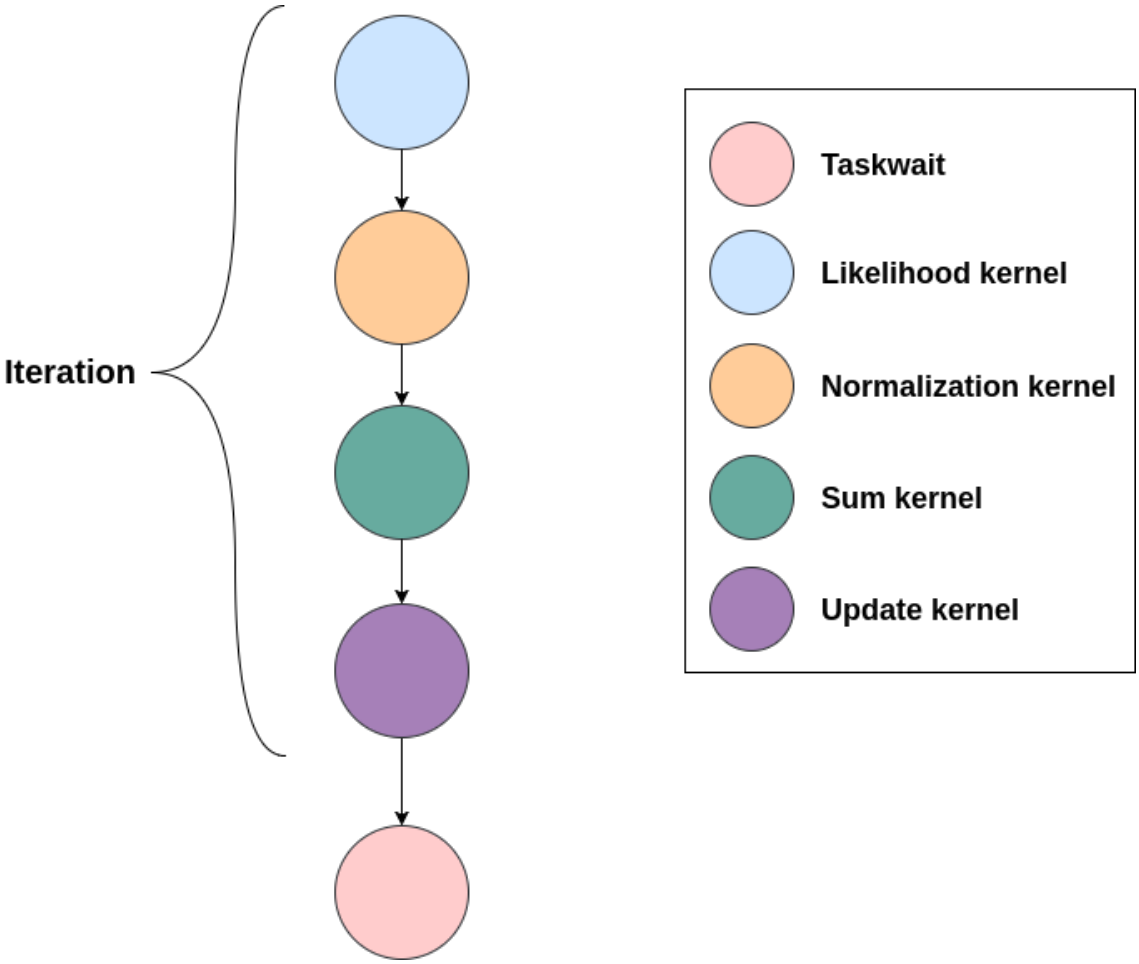

Figure 3.7: Particle Filter task graph

to be move to the GPU and back to the host in each iteration.

This benchmark has been used to test different capacities of the CUDA environment which are not necessarily obvious or are not present in the documentation but are needed to obtain a good performance in the runtime. This included the limit of prefetch operations that can be simultaneously performed on the GPU before the performance degrades, the performance of different synchronization approached as well as the optimal number of streams to use when executing tasks.

The task graph is identical to that of the Multi-Saxpy application previously shown.

Chapter 4

Design & Development

The goal of the Nanos6 project is to redesign and improve upon the previous runtime, thus the choices made when designing the new runtime have been done taking into account the shortcomings of the previous version. This section describes the design of the runtime and explains the reasons behind the decisions made. It is structured in 2 sections, 4.1 describing the existing Nanos6 components involved in detail and 4.2 illustrating the changed made.

4.1 Nanos6 Runtime

4.1.1 Hardware

In order to assign resources and schedule work efficiently, the runtime needs to be aware of the underlying hardware of the system. The hardware is read on start-up and a tree of objects representing the hardware is built on the runtime. This tree is based on two different kind of objects:

Memory Place A memory place is a device on the system in which data can be placed. This refers to the different NUMA nodes on a machine or to the device memories of accelerators. It can also represent user manageable specialized hardware such as the High Bandwidth Memory on a Xeon Phi KNL.

Compute Place A compute place is any kind of place where a task can be run, which comprises CPUs and CUDA GPUs at the moment. Compute places are related to a Memory Place, which corresponds to the ones accessible by them, so for instance, a CPU in a NUMA system contains links to all the NUMA nodes in the system, but not to any separate device memory (such as a GPU memory).

The hardware tree can be accessed by any other element of the runtime, such as the scheduler, in order to optimize the execution of tasks by, for example, exploiting locality. The diagram on figure 4.1 shows the structure of the hardware information component.

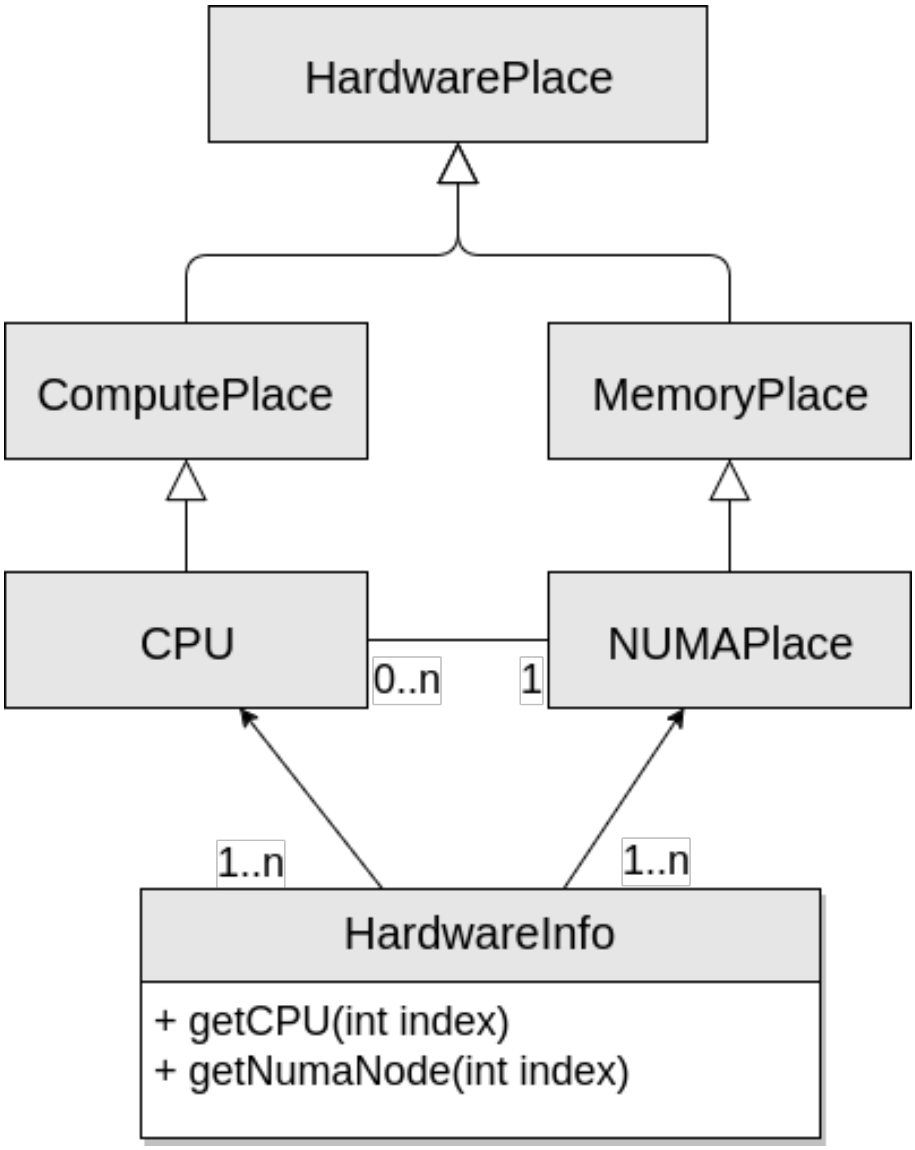

Figure 4.1: Structure of the Hardware Information component

4.1.2 Scheduler

Before describing the mechanisms for the execution of tasks, it is important to understand how the runtime schedules these very tasks for their execution.

The time a task spends *blocked* (as described in section 3.2.1), as well as the time in which it is *finished* are responsibility of the dependency system. The scheduler instead, becomes relevant when tasks become ready.

The scheduler provides an API for the executor components of the runtime. The different threads can query this interface to request ready tasks. The scheduler will in turn, return a task appropriate to the computing resources that asked for the task. In this way, the scheduler acts as a passive component, which keeps track of the ready tasks and waits for the threads to request tasks, at which point it provides an adequate one according to its scheduling policy.

The scheduler is also responsible of managing the different computing resources. What this means is that when a thread requests a task but the scheduler has none to provide, it must mark the thread as idle, and when a ready task is added an idle

thread may be resumed in order to execute the new task. This is required because threads with no work to execute (when there are not enough tasks to use all the CPU or when the load balancing is not perfect) become asleep.

The general design (excluding the components introduced to provide CUDA support) of the scheduler is shown on figure 4.2.

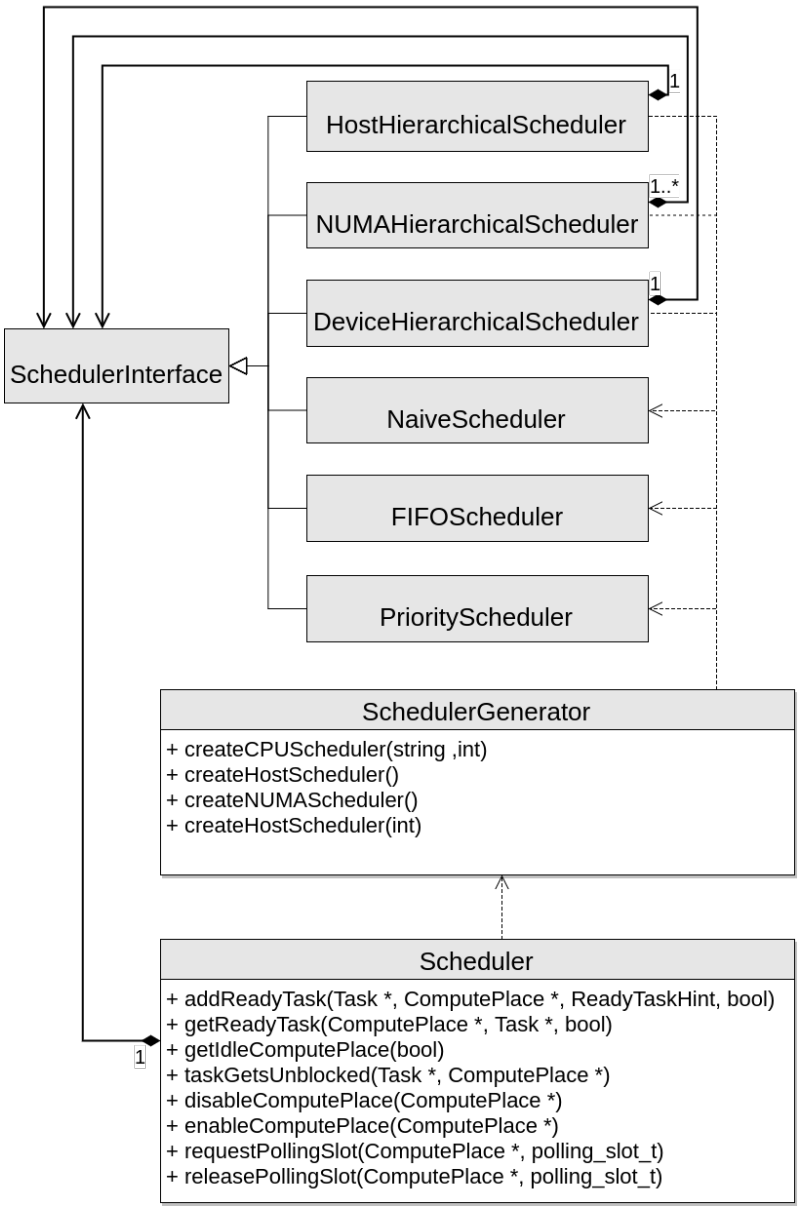

Figure 4.2: General view of the scheduler component of Nanos6

The general design is relatively simple, there is a `Scheduler` class which contains a single instance of a `SchedulerInterface`, which is the actual implementation of the scheduler.The hierarchical nature of the scheduler as described in section 3.2.1, in constructed by using a `HostHierarchicalScheduler` as the base, which contains another instance of a `SchedulerInterface`. Not all schedulers are reflected in the diagram, however, the bottom 3 are actual scheduler implementations and will be at the leaves of the hierarchical tree of schedulers.

The scheduler tree by the `SchedulerGenerator` is built according to two envi-

ronment variables. `NANOS6_SCHEDULER` determines the topmost level of scheduling, it accepts either "hierarchical" to build a hierarchical scheduler or any device scheduler value to use a monolithic scheduler, using one of the low level schedulers as the base. When the scheduler is hierarchical, the variable `NANOS6_DEVICE_SCHEDULER` determines which scheduler is used at the leaves of the tree.

Since the scheduler needs to keep track of the computing resources, all the low level scheduler implementations keep some list of the CPUs as well as a list of ready tasks. How these elements are stored depends on the actual implementation and the policy of the scheduler.

The definition of the API provided by the scheduler is shown in the following snippet of the Scheduler class.

```
1  virtual ComputePlace *addReadyTask(Task *task,
2      ComputePlace *computePlace, ReadyTaskHint hint,
3      bool doGetIdle = true);
4
5  virtual void taskGetsUnblocked(Task *unblockedTask,
6      ComputePlace *computePlace);
7
8  virtual Task *getReadyTask(ComputePlace *computePlace,
9      Task *currentTask = nullptr, bool canMarkAsIdle = true);
10
11 virtual ComputePlace *getIdleComputePlace(bool force=false);
12
13 virtual void disableComputePlace(ComputePlace *computePlace);
14
15 virtual void enableComputePlace(ComputePlace *computePlace);
16
17 virtual bool requestPolling(ComputePlace *computePlace,
18     polling_slot_t *pollingSlot);
19
20 virtual bool releasePolling(ComputePlace *computePlace,
21     polling_slot_t *pollingSlot);
```

Listing 5: Scheduler API declaration

The following is the role of the different functions:

addReadyTask This is the function called by the dependency system to add a task whose dependencies have been resolved. In addition in may be called by the external Nanos6 API when a task with no unresolved dependencies is created.

This function will add the task to the list of tasks on the scheduler and will return an appropriate computing resource (the ComputePlace) from its idle device/thread list, which will be waken up.

taskGetsUnblocked This function is called when a task that was blocked it unblocked. This is not relevant to this project, since only SMP tasks can be blocked and this functionality has not been explored.

getReadyTask This function is called by the different threads or devices in order to obtain a task that is ready to execute. This function receives the calling ComputePlace as a parameter in order to return the adequate task that may be assigned in a previous moment or may be determined when the function is called.

(enable/disable)ComputePlace These functions are placed in the API for future use.

(request/release)Polling Used for schedulers which have a polling mechanism, as a substitute for the `getReadyTask` function.

4.1.3 Executors

The execution of tasks in the runtime is done through the usage of Worker Threads, which represent the underlying execution threads which run the application. One thread is created for each CPU on the system by default, with its corresponding `WorkerThread`. Each of these worker threads executes a loop with the following steps:

1. A task is requested from the scheduler.
2. The *body* of the task (the user level code of the task) is run.
3. The dependency manager is notified of the finalization of the task.

If a ready task is not found the thread will become registered in the schedulers idle list and the thread will become asleep, unless the scheduler used is a polling scheduler, in which case they can stay polling until a task is given. The worker threads can be later resumed up by the scheduler using the `ThreadManager` which manages the state of all the threads on the system. The structure of the executors of the runtime is illustrated on figure 4.3.

At the initialization phase of the runtime, a thread is created for each CPU. From that moment onwards the pausing and resuming of components is done by the `CPUManager` and the `ThreadManager`. Worker threads will not directly communicate with the dependency manager, instead a class `TaskFinalization` offers a function to release the dependencies in order to decouple the executor elements and the dependency manager.

In the same fashion, the Scheduler does not directly handle the threads (although it keep track of the CPUs), but instead delegates the task of handling the computing resources to the respective manager classes. The threads, however, do access the scheduler directly through the interface of the system.

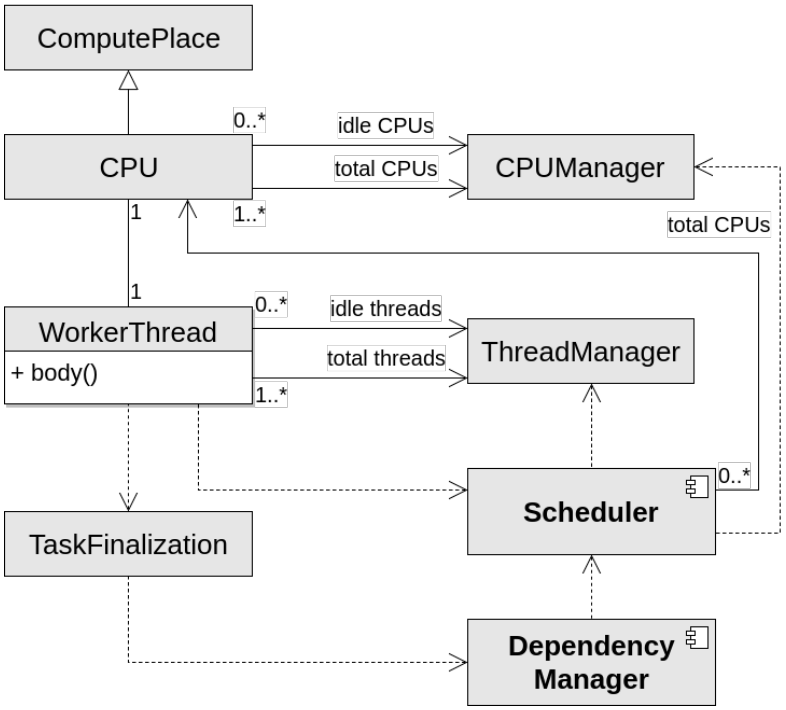

Figure 4.3: General view of the executor component of Nanos6

Task execution

Execution of the SMP tasks is shown in the diagram on figure 4.4.

The life of a task starts when it is created and submitted to the runtime (not depicted on the diagram). When this is done, two things can happen, on the first case the task has no pending dependencies and it is directly added to the scheduler; on the second case there are unresolved dependencies and the task is stuck on the dependency manager until they are solved. Once the dependencies are solved, the dependency subsystem calls the `addReadyTask` function of the scheduler. This may return a computing resource that will be resumed immediately. It must be noted that all these functions are executed on the underlying hardware thread of the `WorkerThread` that released the dependencies of the task on which the added task was waiting.

Once the worker thread is resumed, it will begin running its body in a loop. At the start of each iteration it will call the `getReadyTask` of the scheduler, obtaining a ready task or nothing if no task is available. If a task is obtained, the `body` of the task is executed and at its return, the function `disposeOrUnblockTask` of the `TaskFinalization` is called, which may cause pending tasks to become ready and start the cycle again. If not task is returned by the scheduler, the thread will instead be paused.

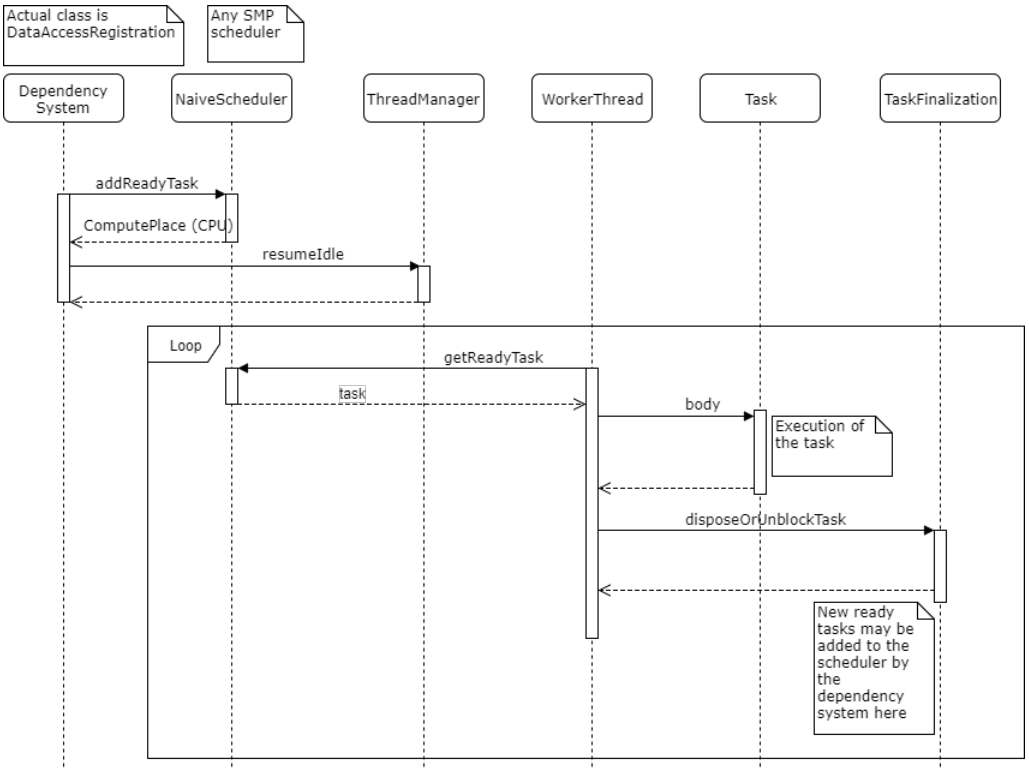

Figure 4.4: Program flow of the execution of a SMP task

4.1.4 Nanos6 API

The Nanos6 API is the external component of the runtime, an interface used by the intermediate code generated by mercurium to communicate with the runtime. The API exposes a number of structures and functions which are needed to use the runtime. While the API is too extensive to list, the most crucial functions and structures for the execution of tasks are described as follows:

`nanos_create_task` This function is used to create a Task object on the runtime, however, it is still pending submission to the dependency manager or the scheduler

`nanos_submit_task` This function is used to submit a task previously created. This is the function that actually registers the task on the runtime so that it may begin its execution.

`nanos_taskwait` This function replaced any annotation `#pragma oss taskwait`, marking where a synchronization point exists. For this, the runtime creates a special task depending on all submitted tasks. In Nanos6 taskwaits are implemented creating a tasks that is blocked by all existing tasks and pausing the main thread which runs the user code until the taskwait is solved.

`nanos_task_info` This is a structure used to keep data needed for the execution of a task, such as the identification of the dependencies of the task and a pointer to the function containing the body of the task.

The most relevant component of the API for this project is the `nanos_task_info` structure. Since it is the container where the data used to execute the task is stored,

it is where modifications have been made to support GPU tasks. The description of this structure is as follows:

```
typedef struct {
        void (*run)(void *args_block,
            nanos6_taskloop_bounds_t *taskloop_bounds);

        //! \brief Function that the runtime calls to retrieve
        //      the information needed to calculate the dependencies
        void (*register_depinfo)(void *handler, void *args_block);

        //! \brief Function that the runtime calls to obtain
        //      a user-specified priority for the task instance
        nanos_priority_t (*get_priority)(void *args_block);

        char const *task_label;

        char const *declaration_source;

        //! \brief Function that the runtime calls to
        //      obtain an estimation of the cost of the task
        size_t (*get_cost)(void *args_block);

        char const *type_identifier;
} nanos_task_info __attribute__((aligned(64)));
```

Listing 6: `nanos_task_info` structure definition

This part of the API is enough to translate a simple annotated task. When a task annotation is found, the compiler will generate a function containing a *create task* and *submit task* function calls, as well as a `nanos_task_info` structure corresponding to the data required for that function. In the same manner when a taskwait annotation is found, it is simply substituted by a `nanos_taskwait` call.

4.2 Nanos6 CUDA changes

4.2.1 Unified Memory

Before describing the changes to the code base, it is important to contextualize the usage of unified memory. While a CUDA implementation would generally require the creation of memory management elements on the runtime, none are needed on the current implementation due to the early decision of using Unified Memory. The previous runtime (Nanos++) required keeping track and moving around of CUDA memory, which translated to the following:

- The runtime allocates a pool of CUDA memory at start-up to use during the execution.
- The runtime keeps a list of memory addresses and the current locations on a directory and keeps track of dirty data.
- The runtime copies the input data marked for copying on a CUDA task before the execution to the GPU. This implies either host to device or device to device copies.
- The runtime copies all the output data of the CUDA tasks back to the host in every Taskwait operation.

This, together with the fact that the synchronization between memory transfers and kernel executions is done in the host side (via registering CUDA events and polling their status) can cause performance issues.

The following list contains the shortcomings found on the usage of the CUDA memory in the Nanos++ runtime:

- Initial allocation of CUDA memory is done by allocating (by default) a 95% of each GPU, which is in most cases unnecessary. This solution did not present a problem originally, since older GPUs had smaller memory sizes, however, on modern devices it can result on a big performance loss due to the allocation of unnecessary memory. It is possible to tune this using environment variables, however, for a inexperienced user it may present a hard to identify source of bad performance. On the other hand, it may not be possible to know beforehand how much memory will be enough.
- The semantics of OmpSs specify that after a taskwait the data on a program should be accessible by the user, which means that the data on the GPU have to be copied after a taskwait. While not usual, this may mean copying of data that is not necessary, for example, output variables on a kernel used only as input on a subsequent kernel. While it is again possible to tune this "flushing" of data via OmpSs pragmas, it can be a source of issues for inexperienced users.
- Tracking the location of data is not done efficiently. Since synchronization between memory operations and kernels is done on the host instead of on the CUDA streams, there are delays introduced between the different device operations. This documents get into more detail regarding the transfer/kernel synchronization on the section dedicated to the execution of tasks.

In addition, using device memory requires tracking where each region of data is stored at any point in time as well as what copies have been made and where are they located currently. In Nanos++ this was done by implementing a global directory of memory regions which contained the location of every region at the current point in time as well as smaller directories with the data on each device, including their status, modifications, etc. Creating this functionality is a difficult and time consuming task.

Considering all of the previous, it was decided to use the Unified Memory mechanism provided by the runtime of CUDA. This provides a series of benefits and drawbacks, and it is enabled only by the new features in Nvidia Pascal GPUs.

Unified Memory was introduced in CUDA versions 6, however, it was not as sophisticated as the most recent version. The following restrictions, found on the first iteration of the mechanism, would make a OmpSs-2 runtime using unified memory impossible:

- Before a kernel is run, the semantics specified that all managed data (data allocated for the unified memory) has to be copied to the GPU. This in itself, is not a problem, however, in practice, the runtime does not only copy the data needed for the kernel, but all of the managed data on the system. This causes unneeded movement of data, as well as limiting the potential for Multi-GPU execution, due to all the data being stored on a single GPU.

- A limitation for all programs before the CUDA 8 version is that the maximum amount of data that can be allocated in the GPU corresponds to the total device memory, which can be insufficient for some applications. This is probably related to the runtime transferring all data to the GPU at the launch of a kernel, but it cannot be said for sure.

- Versions of the runtime previous to CUDA 8 and GPUs not supporting concurrent execution, it is illegal to access any managed data while in the time between a kernel launch and a host side synchronization with the device. While it is reasonable for performance not to access simultaneously data on the GPU and the CPU, the constraint specified that no managed data may be accessed, even if it is not used by the kernel. This severely limits the possibility to overlap CPU and GPU execution, which is one of the strengths of OmpSs.

CUDA 8, introduced together with the Pascal Architecture, relaxes these constraints by introducing the following mechanisms:

- GPU side page faults are introduced. This allows memory transfers to be made on-demand instead of needing to transfer all the managed data before a kernel execution. Previously this mechanism was used in device-to-host transfers, using CPU page faults, and now it is possible to use it also on input transfers.

- The limit on allocated managed data is changed to the maximum virtual memory of the system.

- Concurrent access of data is allowed on devices that support it, though, simultaneous access of the same data in host and device results on significant performance losses.

Alongside this changes, a number of operations allowing for optimizations on unified memory are introduced in CUDA 8, such a prefetch operations a hints for the preferred or accessed locations of different memory regions. In summary, it can be said that the focus of unified memory has moved from being solely a mechanism for fast prototyping to being a memory model from which reasonable performance can be attained.

Considering the development difficulty and inefficiencies that can be introduced by the need to develop manual memory management, and the features on modern Nvidia graphics processors, it was decided to use unified memory on the runtime to

leverage the task of memory management on devices on Nanos6. For this reason, the designs on this chapter, lack references to device memory management, which should be present if manual memory management needed to be implemented, instead the focus of the project has been in the execution and evaluation of the system under unified memory.

4.2.2 Hardware

The only change required by the runtime on the hardware information component is the addition of a `ComputePlace` representing a CUDA GPU. Without unified memory an additional `MemoryPlace` would have to be added, which would exist in a 1 to 1 relationship with the CUDA devices, however, there is no need to do so when using unified memory. A illustration of the changes on the design can be found on figure 4.5

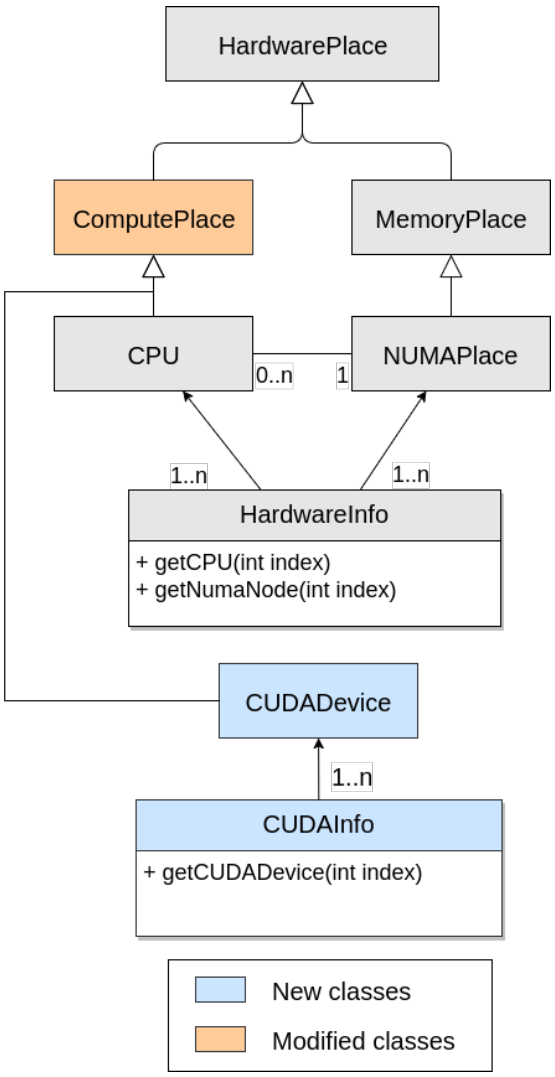

Figure 4.5: Changes on the hardware component of Nanos6

The `CUDADevice` class represents a single GPU on the system. Besides the generic data kept by the generic compute elements, it keeps specific data and functions required for the execution of tasks. The threads will directly call the functions on

the `CUDADevice` to run tasks. More details on this are presented in section 4.2.4.

4.2.3 Scheduler

As described in section 4.1.2 the scheduler of Nanos6 is a hierarchical scheduler. In order to support the scheduling of CUDA tasks a new sub-tree of schedulers need to be developed in parallel to the NUMA scheduler. While the NUMA scheduler keeps a scheduler for each of the memory nodes on the system, the CUDA scheduler only has a single scheduler for all the GPUs on the system. The reason for this is that several CPUs are accessible from each memory node, however, that does not apply to CUDA devices since each device has its own memory. Thus the scheduler which maps to the memory, the corresponding to the `NUMAHierarchicalScheduler`, is the lowest level scheduler. Figure 4.6 shows the changes done to the scheduler component.

The `CUDAScheduler` keeps track of the state of the different GPUs and routes the different CUDA tasks to different devices. Unlike the threads the CPU schedulers track, the `CUDADevices` are idle by default, thus, all the devices need to be added to the scheduler idle list at start up and the `body` function of the device will be called when a CUDA ready task is added to the scheduler. How CUDA tasks are identified is described on section 4.2.5 and details on the execution on section 4.2.4.

In addition to this a number of changes have been made all the schedulers:

- An additional scheduler is created on the `HostHierarchicalScheduler` where the `CUDAScheduler` will be located.
- A check is added to the `addReadyTask` of all scheduler implementations in order to ensure that each scheduler handles its own task type.
- A check is added to the `getReadyTask` of all scheduler implementations in order to ensure that each scheduler handles only appropriate `ComputePlaces`. In addition an attribute is added to the base `ComputePlace` that identifies the type of tasks that it can execute.
- A identifier "cuda" is created to configure the CUDA scheduler and all needed functions to generate it are added to the scheduler generator.

A scheduler created for a system with two NUMA nodes and two GPUs is illustrated on figure 4.7.

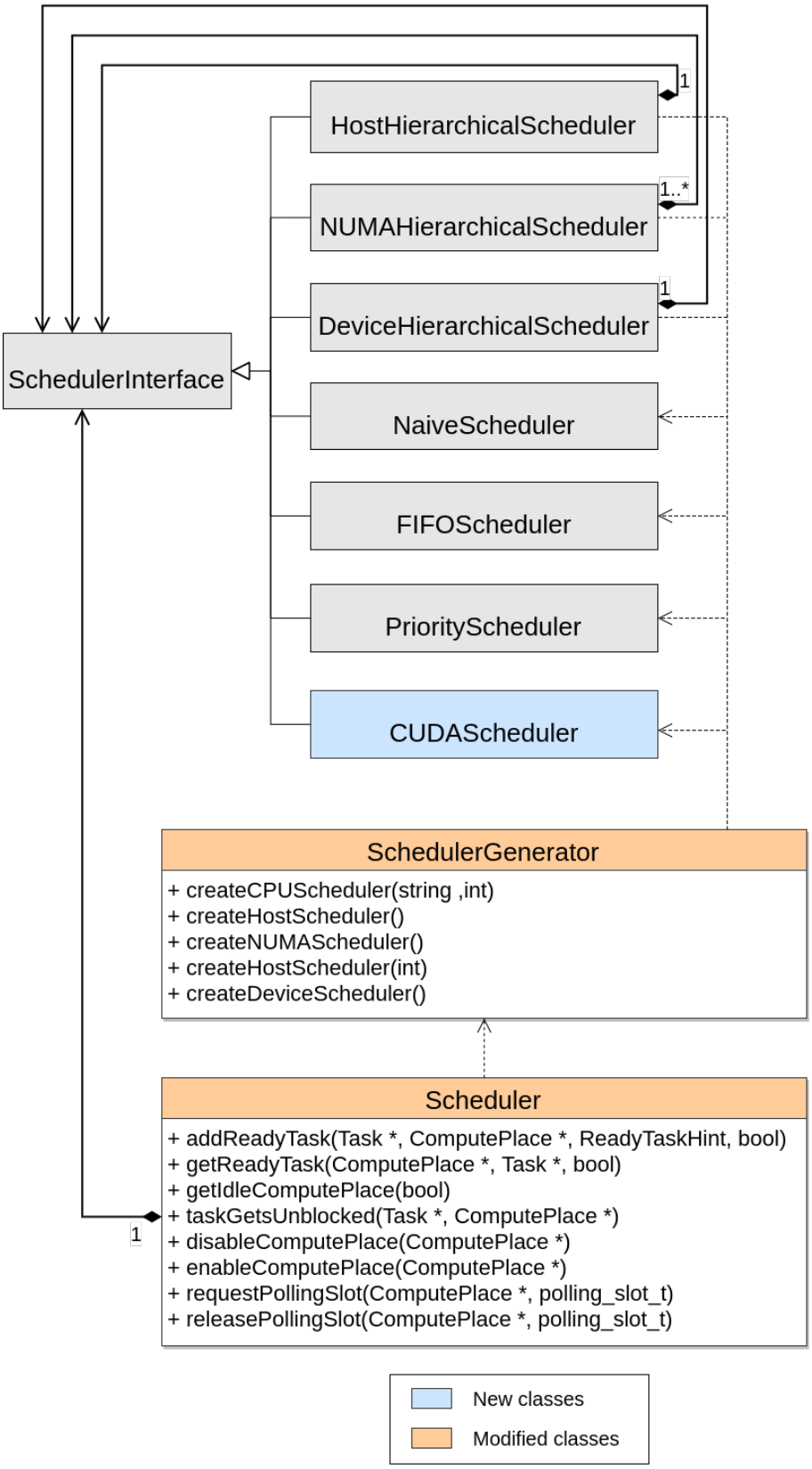

Figure 4.6: Changes on the scheduler component of Nanos6

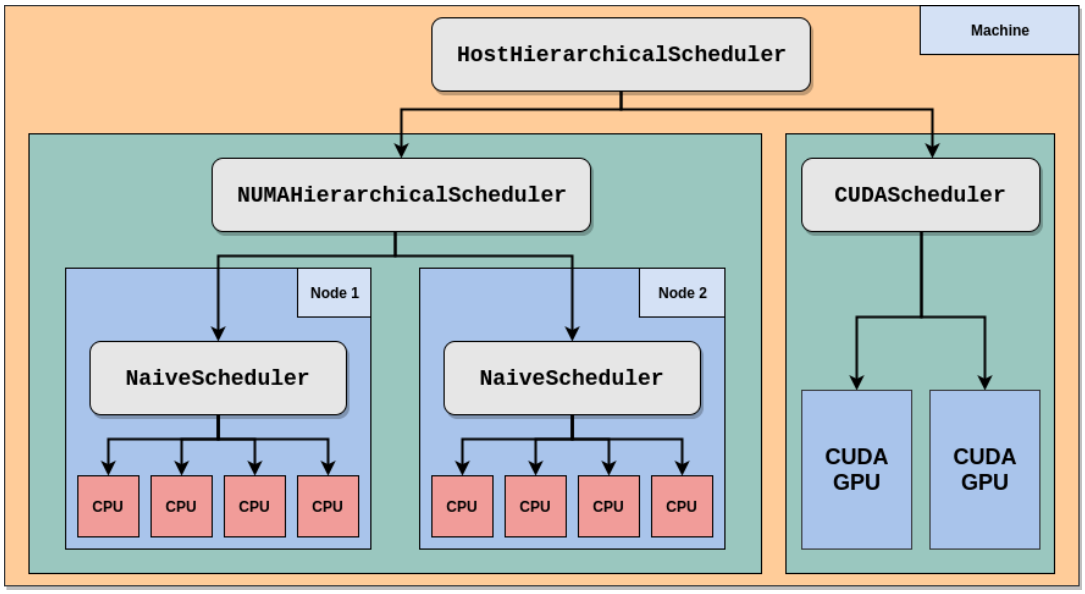

Figure 4.7: Illustration of a hierarchical scheduler with CUDA GPUs

4.2.4 Executors

While other components have kept similar designs in respect to their SMP counterparts, the execution of CUDA tasks is drastically different to the SMP tasks. This is due to their asynchronous nature; while SMP tasks are synchronous and keep the CPU running them occupied while being executed, CUDA tasks only need the host to execute a small launch function after which the CPU is free to execute a different task. In the same manner, the synchronization needed for the task finalization is drastically different; SMP tasks do not need to be synchronized, however, a mechanism needs to be created in order to detect when CUDA functions have finished their execution. The diagram showing the changes on the executors is illustrated in figure 4.8.

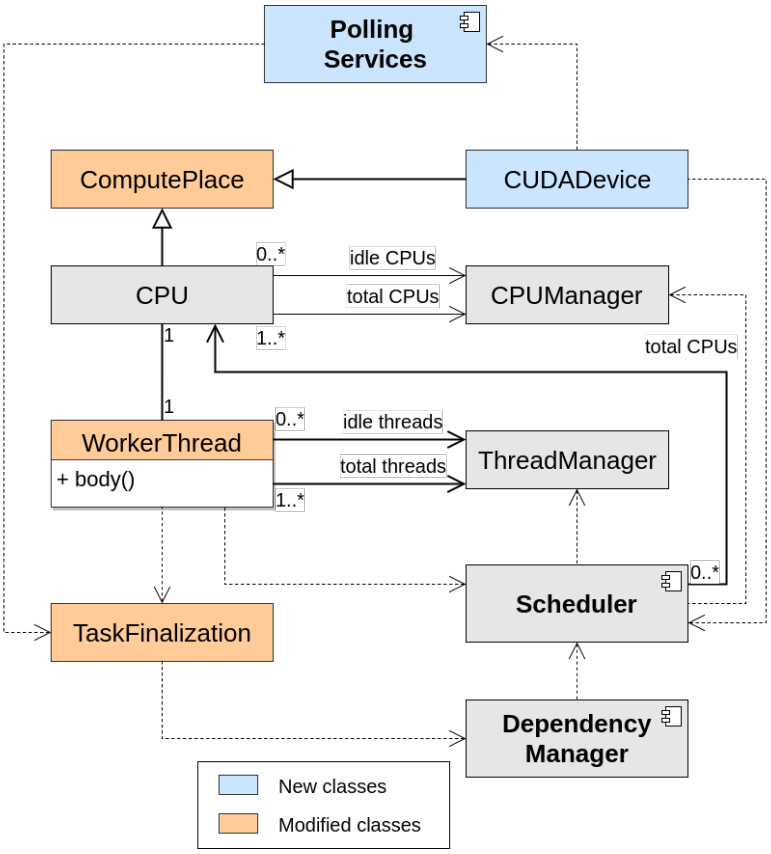

Figure 4.8: Changes done to the executor component of Nanos6

A number of new elements are introduced to the executor component:

`CUDADevice` As described in section 4.2.2, this class is a representation of the physical CUDA processor. It contains a `run` function which requests tasks from the scheduler and launches them, however unlike the `WorkerThreads` which run in a loop until they are idle, the run function is called only once and will attempt to launch as many CUDA tasks as possible before becoming idle again. A description of this process is given in section 4.2.4. In addition the objects keep a pool of streams to be used to launch the tasks on the GPUs; the stream usage policy is detailed in section 4.2.4.

`Polling Services` The Polling Services is a component introduced during the development of this project. The role of this component is to query the different

devices in order to find out when the execution of an asynchronous task has finished executing, although the service is generic enough to be used for other purposes in the future. This component is described in section 4.2.4.

`CUDAManager` The purpose of the manager is to act similar to its counterpart for SMP devices, keeping track and providing functions to manage the devices on the system. However, currently it is only used on the initialization and shutdown of the system.

In addition to this, changes have been made to the different involved components of the runtime:

- A number of changes have been made to the dependency manager, instrumentation, initialization and executor components to remove the assumption that tasks will always be run on a SMP thread.
- Calls to the polling services have been introduced in the *idle* code of the worker threads as well on the loop of the `LeaderThread` (a description of this is given in section 4.2.4).
- Modifications have been done to the API of the system to include the polling services functionality.

Task Execution

A diagram showing the task execution flow of a CUDA task can be found on figure 4.9.

Compared to the graph for SMP task execution (in figure 4.4) there are some key differences.

The first major difference is that the `WorkerThreads` running the SMP tasks are woken up by the `ThreadManager` but they run on another thread. The thread manager is called by the dependency system running on another worker thread or in the main thread. The `CUDADevice` running GPU tasks on the other hand is directly called by the thread that wakes it up, and will run the code on the same thread. This means that when a `WorkerThreads` call the dependency manager and the dependencies of the CUDA task are solved, the function used to launch the tasks is directly called in the same hardware and all tasks are launched to the GPU before returning control to the worker thread. Since the host side of the CUDA tasks has a very small computational time there is very little time in which the worker thread is not running CPU tasks.

This is different from the Nanos++ runtime, in which a *HelperThread* was created for each additional architecture on the system, which would compete for computation time with the SMP threads. Instead in the new runtime, the different architectures cooperate to interleave the execution of different tasks.

Another difference regards the way in which the finalization of the tasks is detected. For SMP tasks, the running `WorkerThread` will simply call the dependency system after the body of the task is run. CUDA tasks require additional functionality in order to avoid blocking the launching thread. CUDA provides two non-blocking mechanisms to do this, CUDA events and CUDA asynchronous callbacks; for this

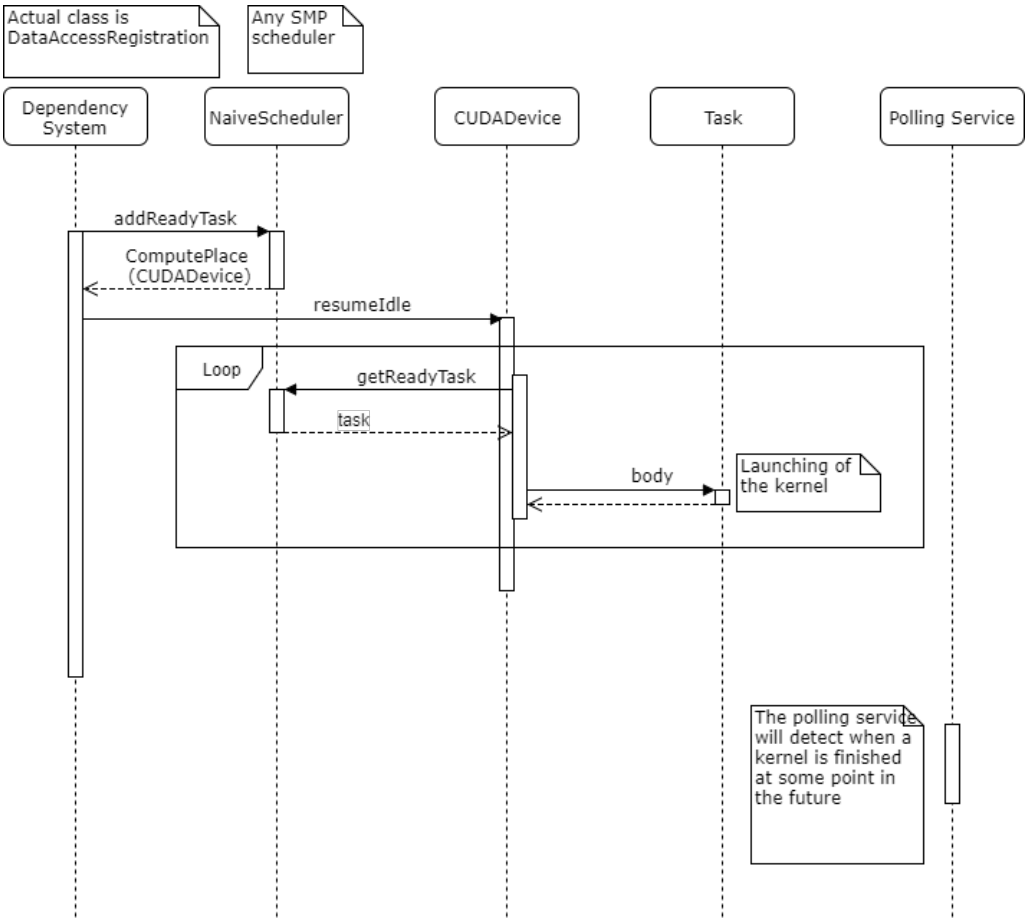

Figure 4.9: Program flow of the execution of a CUDA task

project events were chosen, details are described in the section regarding the polling services (4.2.4).

One last major difference is on how the tasks are run. To run CPU tasks, the worker thread calls the body of the task, solves the dependencies and then asks for another task to run. Running CUDA tasks is fundamentally different; in order to reduce the need for host usage, the system launches as many concurrent tasks as possible and lets the hardware queues of the GPU run them as needed. In order to do this, the `CUDADevice` will request a task from the scheduler, launch it and immediately repeat the process until no more remain. At this point, the device becomes idle again. In order to maximize the potential concurrent execution inside the GPU, different CUDA streams are used to achieve higher parallelism.

In order to improve concurrency in the GPU as much as possible, different CUDA streams are used to execute the kernels.

Streams Usage

To run multiple tasks efficiently, it is essential that a good usage of streams is done. The previous version of the runtime had some issues with its execution model, due to which an effort has been made to address them in the new implementation.

In Nanox, two streams are dedicated to the input and output transfer operations respectively (the `cudaMemcpy` calls). A separate pool of streams is dedicated to the execution of kernels (tasks). This approach provides good concurrency between

task executions, however, it creates several problems. First, it is impossible to have concurrent memory operations on the same direction, due to them being queued to the same stream. Second, since the memory operations and kernels of the same task are queued to different streams, a host side synchronization is needed in order to detect when copies are finished and the kernel can be launched. This host side synchronization is done through the polling of CUDA events queued immediately after the corresponding operation and can introduce a latency between the memory operation and the kernels. An illustration of this mechanism is provided in figure 4.10.

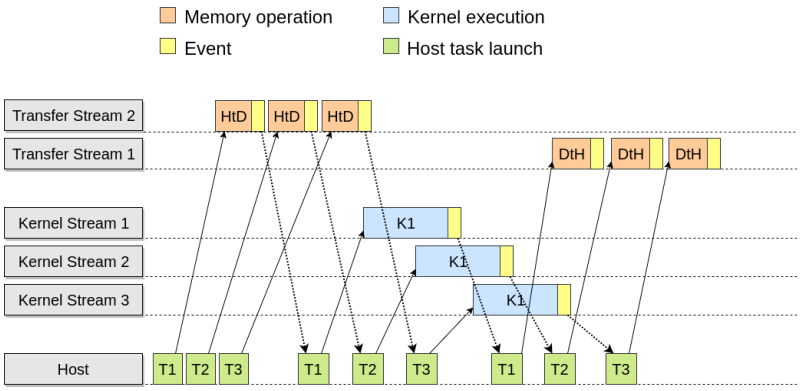

Figure 4.10: Stream usage on Nanos++

This design may have been adequate for older GPUs, however, the restrictions and policies on the usage of streams by the GPU have changed on newer architecture. The requirement of host usage for synchronization between different steps introduces and idle time between different steps of a task execution and takes CPU time.

In order to avoid this, the design of stream usage in Nanos6 attempts to use the natural synchronization CUDA provides through its streams. In order to do this, memory operations and kernels are queued in the same stream and are executed one after the other, removing the requirement of a host as intermediary. For this, a different stream is assigned to each concurrent task, with no upper limit. This is possible to do because CUDA devices have a hardware limit on streams and if the software uses more streams they are mapped to the hardware and no performance penalty is incurred. Thus, by having conceptually unlimited streams of execution, no dependencies are introduced between potentially concurrent tasks. Figure 4.11 illustrates the stream usage policy for Nanos6.

Unified Memory is used in the runtime, so no memory transferences are needed, however, they are still considered in this design. This is because in the future it may be possible to introduce a version of the runtime using regular memory or to add prefetch operations using the same mechanism.

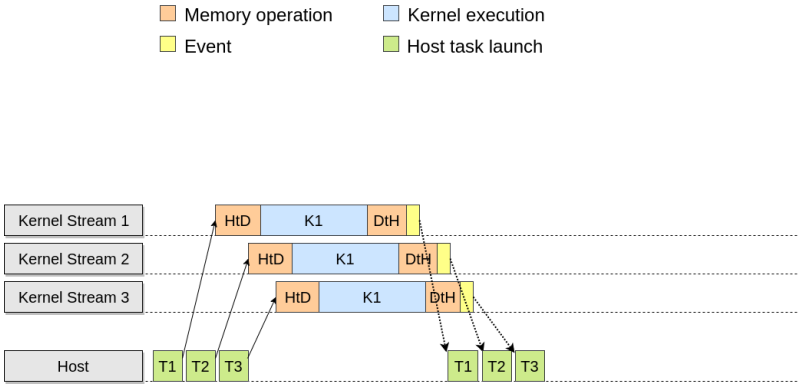

Figure 4.11: Stream usage on Nanos6

Polling Services

The Polling Services is the component in charge of synchronizing asynchronous tasks via polling. This piece of the system is formed by a checking mechanism called by the different SMP threads when idle or before executing a task, as well as an API to register the the data which has to be polled, which in the case of GPU tasks is CUDA events.

A event based synchronization mechanisms was chosen over a callback mechanism due to the response times of both mechanisms. The issue with CUDA callbacks is that they are not called immediately when they can be resolved but they can have a delay, possibly due to their implementation. This is not an issue for events, which on the other hand require an active polling mechanism. Polling however should not be an issue for the system, since the task is performed in the idle time of the Worker threads, and the querying functions have very low computation time, not interfering with the performance of the system.

The API of the polling component is formed of the following two functions:

```
1  typedef int (*nanos_polling_service_t)(void *service_data);
2
3  void nanos_register_polling_service(char const *service_name,
4      nanos_polling_service_t service_function, void *service_data);
5
6  void nanos_unregister_polling_service(char const *service_name,
7      nanos_polling_service_t service_function, void *service_data);
```

Listing 7: Polling Services API

The interface consists of the definition of a `nanos_polling_service_t` function which describes how a polling service is represented. A polling service is a single function that receives some service specific data and returns an integer. This integer represents whether the service has to be removed from the service list to be run after executing its execution. To compliment this definition a couple of functions to register and unregister services are added.

The system is built very to be as general as possible since in the future it will be used by other asynchronous task mechanisms. In order to have synchronization functionality for CUDA tasks, a single service is created which keeps track of the events that have been created through the execution of the program and periodically polls them.

A diagram of the component is shown on figure 4.12.

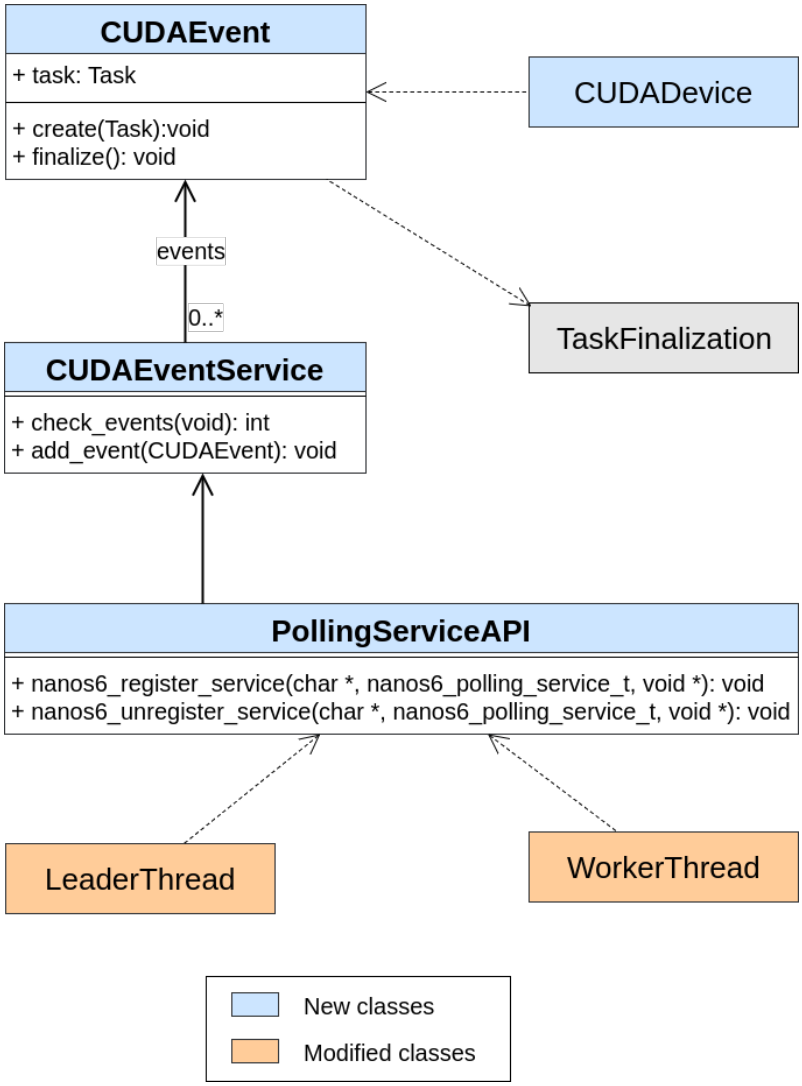

Figure 4.12: A general view diagram of the Polling Services on Nanos6

The interface to call the components is the `PollingAPI` which is called by the worker threads or a leader thread. A function in this API is called to execute the polling function of all the registered services.

This call is done on the worker threads when they become idle. It is implemented by calling the polling just before asking for a SMP task, in order to avoid starving asynchronous tasks when running long CPU code. The reason for the existence of a `LeaderThread` is that the worker threads eventually go to sleep when there are no more ready SMP tasks. This may lead to a deadlock, since an asynchronous task waiting to be detected by the polling services may not be detected due to the worker threads all being stopped. To avoid this, a *leader thread* exists, whose sole role is to call the polling services, and other helper tasks if needed in the future, with a

relatively low frequency. This allows for calls to the polling services even when the worker threads are asleep, without taking up much CPU time when SMP tasks are running.

Since the polling component is designed with the possibility of adding different asynchronous task types in the future, a specific CUDA service must be created. As previously mentioned, the only requirement for a service is to have a function receiving a void pointer and returning an integer.

The `CUDAEventService` is the polling service used to poll the CUDA events generated by the tasks. This is implemented by keeping a list of `CUDAEvents`, objects containing the actual `cudaEvent_t` object as well as a pointer to the task, used on the calls to the dependency manager.

The polling function, simply calls the `cudaEventQuery` function on each event, which will return the status of the events. When the status of a event is successful it will be removed from the list and after the polling process ends, its dependencies will be solved. If the status of the event is caused by an error, the program will be terminated, as there is no recovery function at the moment.

4.2.5 Nanos6 API

The external API nanos6 provides has been modified to accommodate the changes done to the rest of the runtime. A task annotated to be executed on a GPU takes the following form, as shown in listing 8.

```
1 #pragma oss task in([n]x) inout([n]y) device(cuda) ndrange(1, n, 128)
2 void saxpy(int n, float a, float* x, float* y);
```

Listing 8: Annotated OmpSs-2 CUDA task

The requirement for a CUDA task introduces a number of needs for the API. In addition to compiling with the Nvidia CUDA Compiler and the requirement of structuring compiled CUDA tasks in separate files, there are two main issues need to be tackled on the API:

- A way of distinguish between SMP and CUDA tasks needs to be created.
- A way of transferring the stream in which the kernel will be launched needs to be developed. This cannot be done from the runtime since the streams need to be specified in the kernel launch, which is done in user level code and thus need to be generated by the compiler.

In order to identify CUDA tasks a enumerator data type has been declared in the API to identify the type of task. This data type called `nanos6_task_type` is located both in the `nanos_task_info` structures and in the `ComputePlaces` in the runtime. This way, the scheduler can route data to the appropriate device by matching the types of the task and computing resource. The reason to use a enumerator instead of a boolean value is to extend the mechanism easily to any other device type that is developed in the future.

```
enum nanos6_task_type {
        smp = 0,
        cuda = 1,
        type_count = 2
};

typedef enum nanos6_task_type nanos6_task_type;
```

Listing 9: Declaration of the task type data type

The second modification is done to pass the streams to the kernel launch. In order to do this, the definition of the function type used for the body of the body of the function is modified. A void pointer generic parameter is added to the function interface which is designated as a extra parameter. This new parameter will contain data specific to the architecture in use which in the case of the CUDA architecture is the stream. The structure of this data is predefined for each architecture, thus it is responsibility of the compiler to decode and use it.

By adding this extra parameter, the stream decided at execution time can be passed from the runtime to the user level function. The code for a intermediate code CUDA function can be seen in the following code snippet.

```
// Annotated version
#pragma oss task in([n]x) inout([n]y) device(cuda) ndrange(1, n, 128)
void saxpy(int n, float a, float* x, float* y);

// Compiled CUDA function (pseudo code)
void compiled_kernel_function(int arg0, float arg1, float* arg2,
        float* arg3, int ndrange_param_1, int ndrange_param_2,
        void *stream)
{
    saxpy<<<ceil(ndrange_param_1/ndrange_param_2), ndrange_param_2,
         0, (cudaStream_t) stream>>>(arg0, arg1, arg2, arg3);
}
```

Listing 10: Compiled CUDA task in Nanos6 intermediate code

Chapter 5

Evaluation

The following chapter presents the evaluations performed throughout the project, including the analysis of the results and how they have influenced the decision making of the project.

Two different evaluations are presented:

- An initial analysis of the behaviour of the first working version of Nanos6 as well as a couple of evaluations of certain features using the microbenchmark.
- A final analysis of the performance of the current version using all the ported applications.

A series of plots are shown on this chapter of the results of executions of applications and benchmarks. All of the applications contain 2 warm-up iterations in order to reduce initialization anomalies. Moreover, the results are averaged through 10 executions to eliminate other execution anomalies and factors that cannot be accounted for. For the Rodinia applications, the default parameters offered by the benchmarks are used for the analysis, while for the analysis of BSC applications bigger than default datasets have been used to analyze.

Two platforms have been available through the project, all the results presented have been obtained in the second one, a BSC cluster named CTE-POWER, however, for the initial testing and the evaluation of the first working version, a different platform was used. This is due to availability reasons, since the pascal GPUs on CTE-POWER weren't ready for use until later on the project.

5.1 Evaluation Platforms

5.1.1 Ironman

Ironman is the nickname of one of the GPU computing focused machines of the Operating Systems Group on the UPC available and managed by the BSC GPU group. The characteristics of the machine are the following:

- 1 CPU Intel(R) Core(TM) i7-5930K CPU @ 3.50GHz with 12 HW threads
- 4 NVIDIA Corporation GK110BGL [Tesla K40c] GPUs

During this project a NVIDIA Pascal Titan X device was obtained and was placed in the machine replacing one of the K40c GPUs.

5.1.2 CTE-POWER

CTE-POWER[13] is a cluster based on CTE IBM Power8+ processors, with a Linux operating system and Infiniband connection network. The cluster is configured to have 1 login node and 6 compute nodes with the following characteristics:

- 2x IBM PowerNV 8335-GTB @ 4.00GHz (10 cores and 8 threads/core, total 160 threads per node)
- 256 GB of main memory distributed in 32 dimms x 8GB @ 2400MHz
- 2x 480GB SSD as local storage
- 1.6TB NVMe
- 2x nVidia Pascal P100 GPU with 16GB of memory.
- Dual Port Mellanox EDR
- GPFS via two fiber links 10 GBit

The operating system is a Red Hat Enterprise Linux Server 7.3 (Maipo). For the compilation of applications, GCC version 4.8.5 and CUDA version 8.0 have been used. In order to queue applications to the compute nodes SLURM[2] is provided by the platform.

5.2 Microbenchmark Analysis

The first analysis is focused on the first version developed for the application. There are some key differences between the version subject to this evaluation and the version presented on chapter 4. There are two main differences:

- An optimization mechanism not present on the current version existed on this version, using prefetch operations before the launching of the task kernels. Compared to the diagrams for the stream execution model in section 4.2.4 (figures 4.10 and 4.11) a diagram using prefetch operations instead of regular memory copy operations would model the behaviour of this system.

  The reason to use prefetch operations (`cudaMemPrefetchAsync`) is because the CUDA Unified Memory from Pascal architecture onwards transfers memory from host to device and back via page faults. This means that each time there is a page fault on the GPU an interrupt is triggered and the host needs to send data to the device, which takes considerably more time than a regular memory copy. By using prefetches, it is possible to avoid this shortcoming since there is no need for interruptions or host involvement beyond the transfer.

  This would make the runtime work similar to a version having regular memory, still keeping the advantages of Unified Memory and not needing to develop a memory directory. The prefetch operations are, however, not present on the current version due to issues related to their performance, explained in this chapter.

- Unlike the current version where a distinct stream is given to each task, the initial version used a fixed pool of streams which were assigned to each task in a round robin fashion. This introduces some unneeded dependencies between the GPU tasks depending on the amount of concurrent tasks and the pool size, so in a design phase it was replaced by the current version.

5.2.1 Analysis of Performance

A initial analysis of performance was done on the *Nbody* and *Saxpy* applications. The performance of Nanos6 on the Nbody application when compared to that of Nanos++ are almost identical, since Nbody is a very compute focused application and there are little memory transfers involved; making the differences in memory management between both versions negligible.

The Saxpy application on the other hand, revealed some concerning performance issues. The results of the executions of Saxpy programs is shown on figure 5.1. This graph shows the execution of the Saxpy program using Nanos++ and Nanos6 keeping a fixed problem size of 2 GB and dividing the execution into different amounts of tasks.

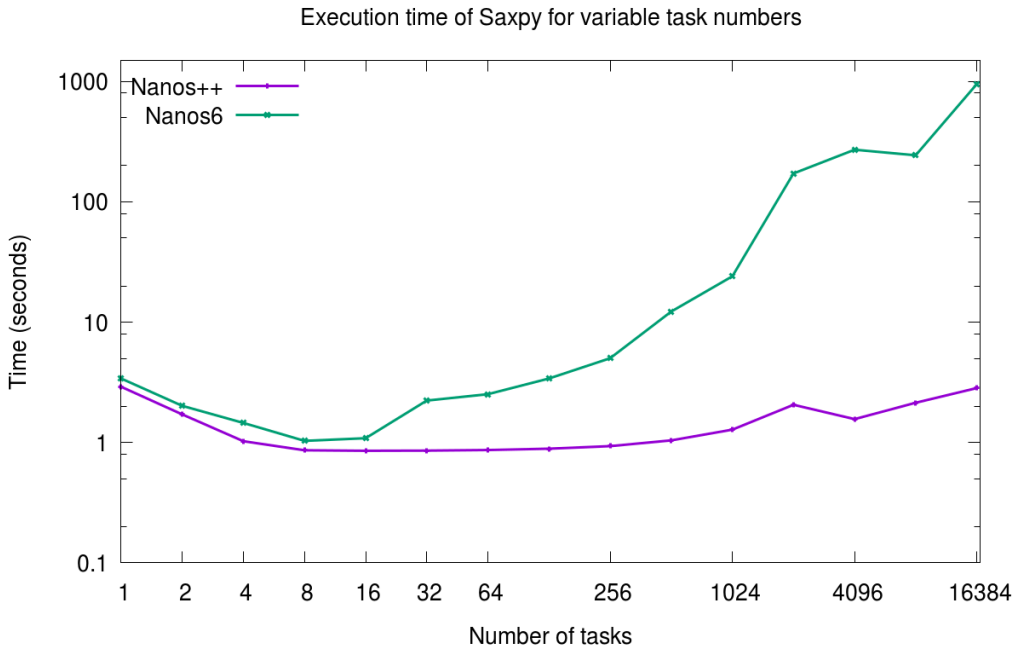

Figure 5.1: Execution time of the Saxpy program with different Nanos versions

Since the amount of tasks increases while keeping the size constant in this analysis, the tasks become smaller and use smaller data. What this translates to is that more kernels are executed and more memory transfers are made. In CUDA it is preferred to perform few big memory transfers over many small ones, so a performance loss is expected when increasing the number of tasks. The execution time increase rate in the application, however, is alarming; due to the scale of the graphic it cannot be appreciated that Nanos++ has a small execution time increase as the amount of tasks increase, however, it is negligible compared to the performance loss on Nanos6.

Taking a look at a profiling of an execution with many tasks, as shown in figure 5.2 the problem becomes apparent. In the section showing the timeline of the host to device transfers in the runtime, it can be observed that the whole region is deep blue. The Nvidia profiler marks sections where transfers are made with different colors depending on the "density" of the transference, that is, the amount of data moved per second. Having a deep blue color means that very few data is being moved in big time intervals.

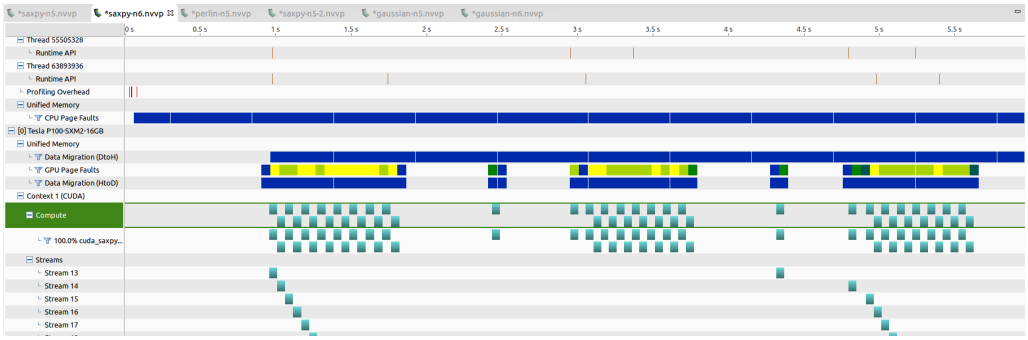

Figure 5.2: Profile of the execution of a Saxpy application

A detailed look at one of the sections shows the problem at hand. One sample, picked at random, shows that the transfer function took 17.629 seconds while the actual data movement took 7,279 milliseconds. Thus there is an obvious issue with the memory management. Hypothesizing that the issue may be on the prefetch operations, since it one of the very few factors affecting memory operations, the execution was repeated using 4 execution modes:

- Using Nanos++.
- Using Nanos6 removing all prefetch operations.
- Using Nanos6 with prefetch operations to copy input data of tasks.
- Using Nanos6 with prefetch operations in both input and output data.

The results of this execution can be seen on figure 5.3.

A diagram focused on the lower time ranges with a linear scale is provided in figure 5.4, in order to show where the performance begins to degrade.

With this graphic available is becomes obvious that the problem is on the prefetch operation usage, however, there is no indication on the CUDA documentation to any kind of limit or any issue with the usage of prefetches. In order to find the problem, a Microbenchmark was developed to have a higher degree of control over the execution.

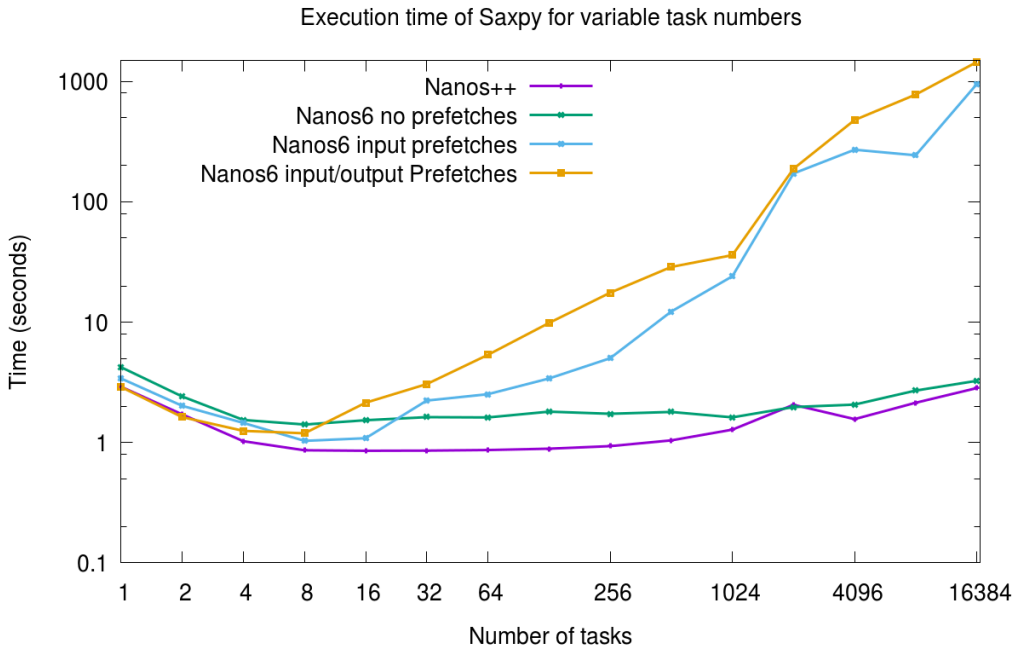

Figure 5.3: Execution time of the Saxpy program with different Nanos versions and prefetch configurations

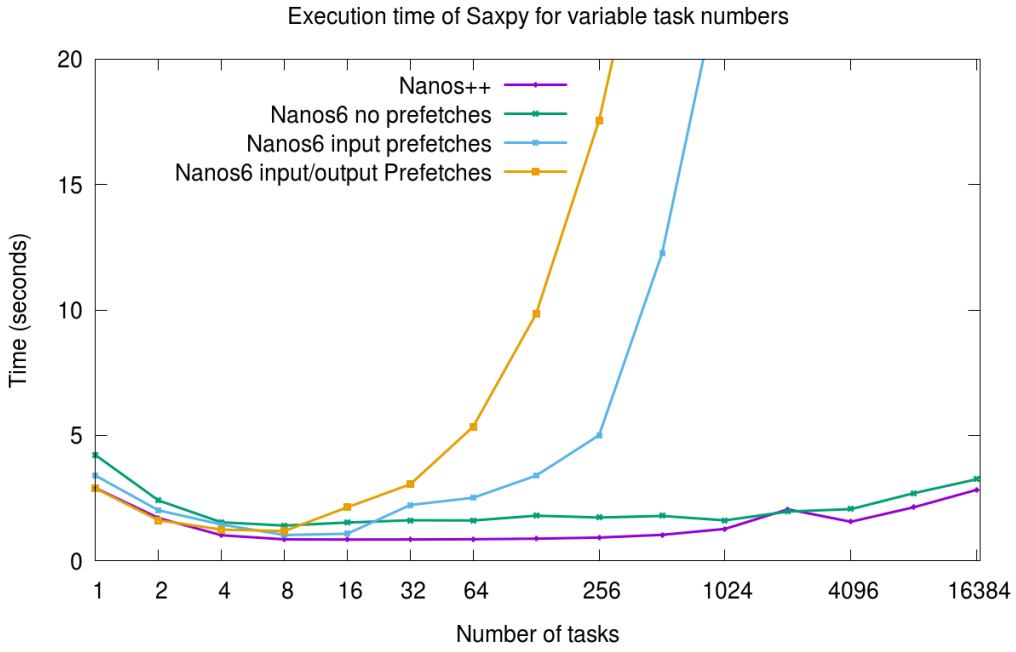

Figure 5.4: Execution time of the Saxpy program with different Nanos versions and prefetch configurations (focused on the Y axis range for small task numbers)

5.2.2 Microbenchmark Analysis

Prefetch Analysis

The scale of the graphics shown up to now is too big to appreciate the differences in performance at smaller numbers of tasks. A hypothesis on the functioning of the performance issues can be developed looking more in detail to the plot in figure 5.4. The idea that there may be a underlying hardware queue for the prefetch operations was suggested by a expert on GPU hardware of the BSC GPU group, which can be contrasted with the results where it can be seen that around 32-64 tasks performance starts to slightly degrade in the versions with prefetch operations.

In order to test this hypothesis the microbenchmark was used to measure the CPU time taken to run a prefetch operation. In order to do this, the time taken to return from the prefetch operation is measured before every kernel launch, keeping a global counter of how many prefetches have been made. The results are shown in figure 5.5.

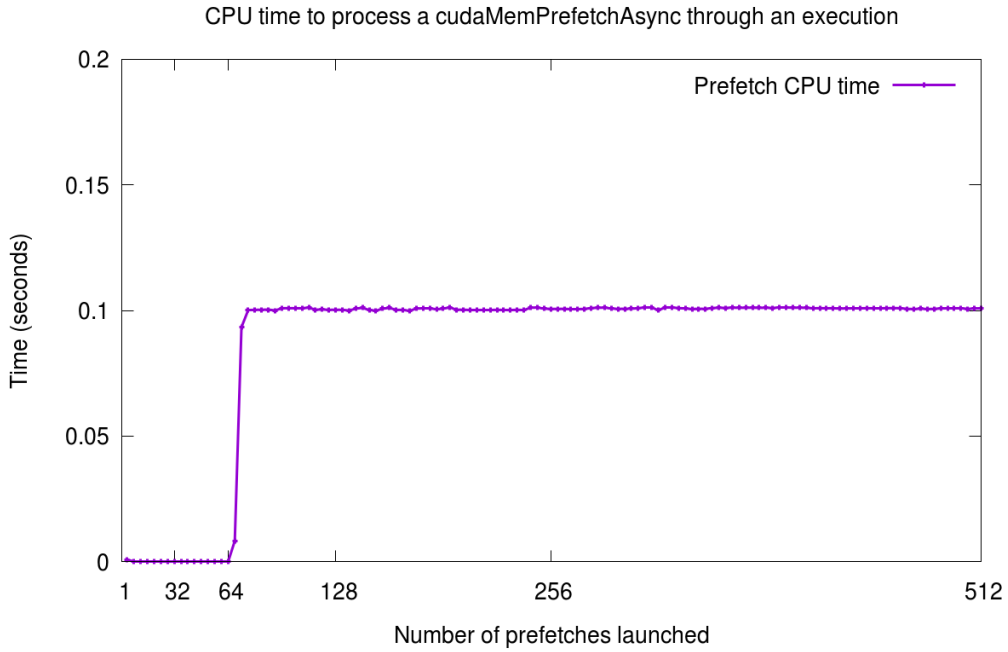

Figure 5.5: CPU Execution time of the `cudaMemPrefetchAsync` operations over a program execution

Something that can clearly be seen here is that the performance starts degrading at 64 prefetches on the device. The reason why Nanos execution starts degrading before 64 tasks, is because each task makes multiple transfers, one per variable in the dependencies.

If as hypothesized there is a hidden hardware queue, it would stand to reason that after the queue having emptied the performance would be back to normal. To test this the program is run iteratively various times and synchronized on the host with a `cudaDeviceSynchronize`, that will wait for all pending operations to finish on the GPU. The results can be found in the graph on figure 5.6.

It can be appreciated where the synchronization points are, since the hardware queue would become empty after the synchronization point and the performance of

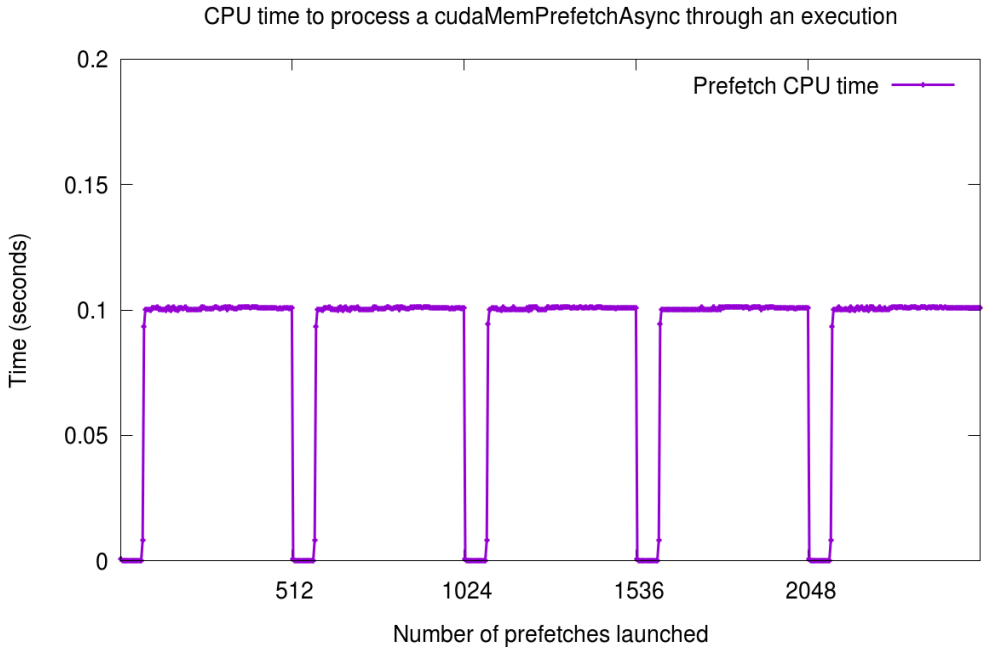

Figure 5.6: CPU Execution time of the `cudaMemPrefetchAsync` operations over a program execution, synchronizing every 512 operations

the next few calls would be good. Afterwards, the queues would become filled until performance began to degrade again. It is possible to see this cycle repeat 5 times, which corresponds to the 5 iterations that have been executed.

There is no documentation regarding the limits on this operations given by Nvidia, thus it is impossible to obtain a certain confirmation. However, this analysis shows that when the performance degrades, the `cudaMemPrefetchAsync` function takes a time to return control to the main program, when the function is supposedly asynchronous and should take almost no time to finish. This could explain the phenomenon observed in the previous section, where the time to process the function was on the order of seconds while the actual time moving data on milliseconds, which could be caused because the function is stuck on the CPU, perhaps waiting for the transfer queue to empty.

This would lead to the conclusion that the limit in the amount of simultaneous prefetches that can be run leads to performance losses on Nanos6. Due to this, the decision was taken to remove prefetch operations and focus on implementing any remaining feature needed as well as focusing on other potential optimizations. Prefetches, however, are not discarded from a future version, but will definitely require some usage policy to avoid this issue.

Before closing the analysis, it is important to note that the argument could be made that most applications would have no problem with this issue, since the amount of GPU tasks tends to be relatively small. In fact, the likely reason why this is not mentioned in the documentation is because it is possible to copy data in batches instead of in many small prefetch operations, which avoids this issue in regular CUDA programs. The problem is that a Nanos6 need to copy the data on-demand and thus cannot copy data bigger than that on the dependencies of the task. While the performance may be acceptable for many applications, the performance

loss is too big on applications with a high task count to consider keeping.

5.3 Performance evaluation

Considering the results of the previous analysis a number of corrections have been applied to the software and a final analysis of performance has been performed. This analysis uses all the applications ported, as described in section 3.2.2. The reasons to use applications from different sources is to obtain data from different benchmark types, for instance, the BSC applications are mostly iterative applications launching a several parallel kernels in each iteration, while Rodinia applications mostly execute their kernels sequentially.

There are many sources of GPU benchmarks that could be used for this evaluation, only Rodinia has been used due to time constraints, since it is required to port each application to OmpSs and OmpSs-2. In fact, only a subset of Rodinia benchmarks has been ported, again due to time limitations, plans for expanding the evaluation suite are mentioned in chapter 6.

A plot with the results of the analysis is presented in figure 5.7. This graph shows the speedup (or slowdown) of the Nanos6 runtime over the Nanox runtime on the different applications ported for this project.

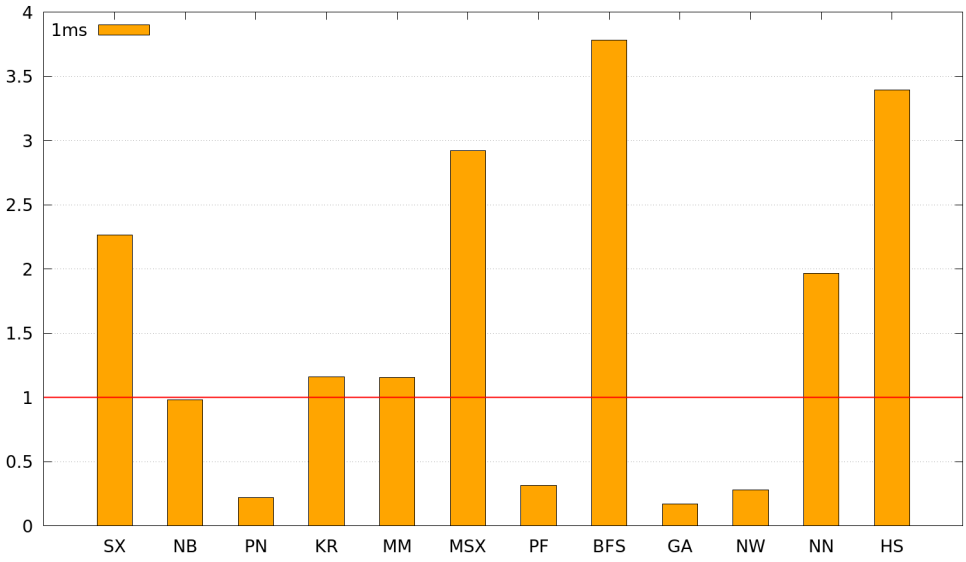

Figure 5.7: Speedup of the different benchmarks

Since there is a regular CUDA version available for the Rodinia applications, another plot is provided comparing the speedup of the Nanos5 and Nanos6 versions over the original program in figure 5.8.

The results obtained are varied, having several applications performing worse in the new runtime, other performing better, and some with no significant speedup or slowdown. However, there are some factors to further analyze for this evaluation.

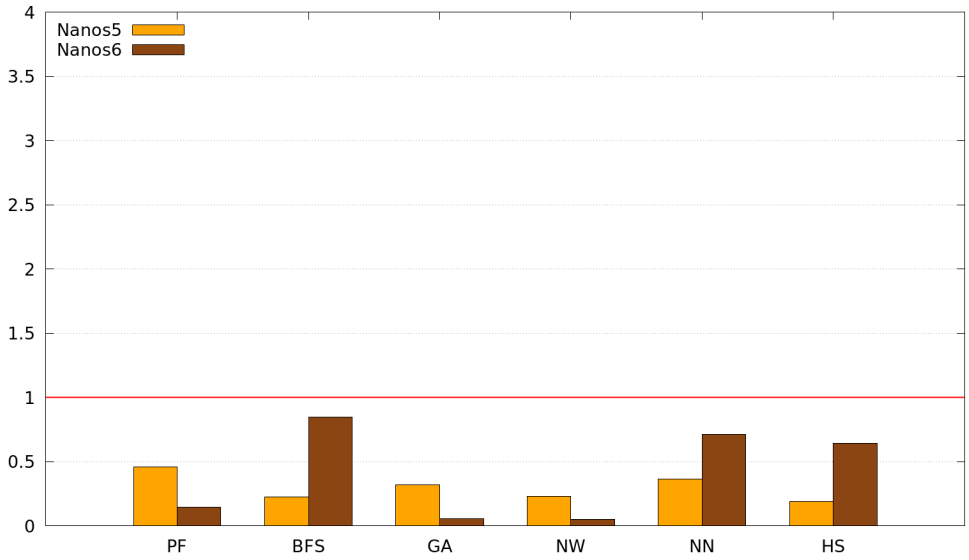

Figure 5.8: Speedup of the different benchmarks using a non-OmpSs version as baseline

5.3.1 Rodinia application performance

The Rodinia applications used on the suite are mostly performance bound, so it is to be expected that they perform better in the new runtime, since the implementation avoids many unnecessary host side computation that could be found on OmpSs.

There are however some applications showing very significant slowdowns, more specifically Pathfinder (PF), Gaussian (GA) and Needleman-Wunsch (NW). Analyzing these applications with the Nvidia Visual Profiler tool, a particular phenomenon can be observed.

These 3 applications have a common characteristic, which is that they execute a huge number of kernels in succession (instead of concurrently), without need for host side synchronization or memory transfers between the kernels. In this cases it can be observed that the space of time between kernel executions is significantly bigger in Nanos6. A small segment of executions as seen in the profiler is shown on figure 5.9.

The reason for this is due to the polling performed by the `LeaderThread` (explained in 4.2.4). This mechanism operates at a rate of 1000 Hz, which corresponds to a query every millisecond. Normally most of the work would be done by the *WorkerThreads* in their idle time, and the leader would only be needed to avoid starving the GPU when there are long running SMP tasks.

These particular benchmarks, however, are only formed by CUDA tasks, with no CPU usage beyond initialization. Due to this, the different threads do not have tasks to run and become idle. Since all of the workers become idle, the responsibility of the polling falls on the leader alone which causes a performance loss due to its low polling frequency. This introduces a delay of up to 1 millisecond between the execution of dependant tasks. In contrast Nanox uses a dedicated Helper thread polling at maximum possible frequency, which reduces the delay between tasks.

In order to demonstrate this hypothesis, the performance of the applications is analyzed with different LeaderThread polling frequencies, from 100 milliseconds to a few nanoseconds. The results of the analysis are shown in figure 5.10 in a plot

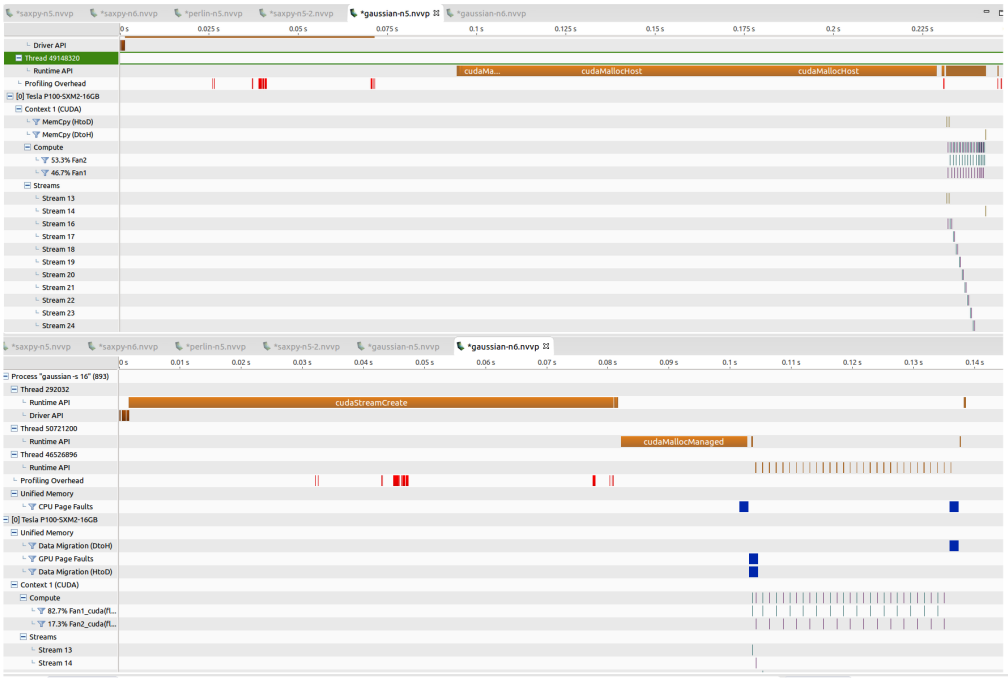

Figure 5.9: Comparison of the execution profiles obtained for different OmpSs executions on the Gaussian application (Nanos++ top, Nanos6 bottom)

showing the speedup of the Nanos6 execution of the application compared to the Nanos++ execution.

First observation to note from the plot is that the Rodinia application with poor performance get significant speedups as the polling frequency is increased. In most cases it is not necessary to maximize the frequency to obtain optimal results, however, when there are no SMP tasks using CPUs there is no reason not to.

Discussing this issue with other Nanos developers it was suggested to use a different scheduler to improve performance without modifying the `LeaderThread`. The reason for this is that there are several scheduler implementation which implement task polling. This mechanism allows WorkerThreads to used a polling mechanism over the scheduler instead of querying it for tasks and pausing if none is found. This way, it is guaranteed that at least one thread will not become asleep and will both poll for tasks and call the Polling services for CUDA. Since the idle loop of the Workers where the polling services are called operates at maximum frequency, an improvement in performance is expected.

The results of comparing the default scheduler with a polling SMP scheduler are shown in the plot in figure 5.11. A comparison using a pure CUDA version as baseline for the Rodinia benchmarks is provided in figure 5.12.

The performance obtained using polling schedulers is similar to that obtained with maximum `LeaderThread` polling frequency, so it works as a temporary solution. It may not be the best solution since it makes the performance of CUDA applications dependent on the CPU scheduler used, however, a definitive solution is left for future work.

The performance of the rest of the Rodinia applications is explained at the end of the chapter, due to them having common features with the BSC applications.

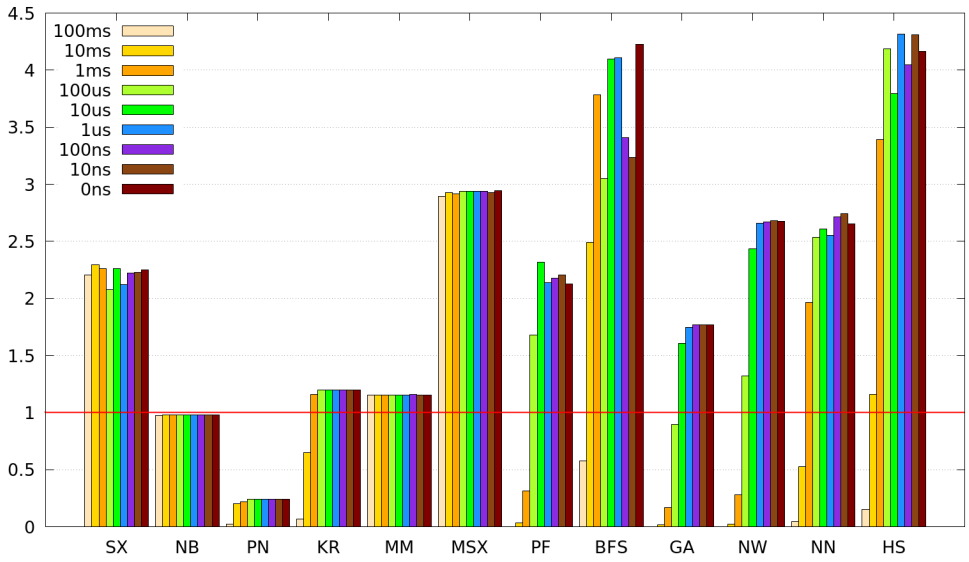

Figure 5.10: Speedup of the different benchmarks at different polling frequencies

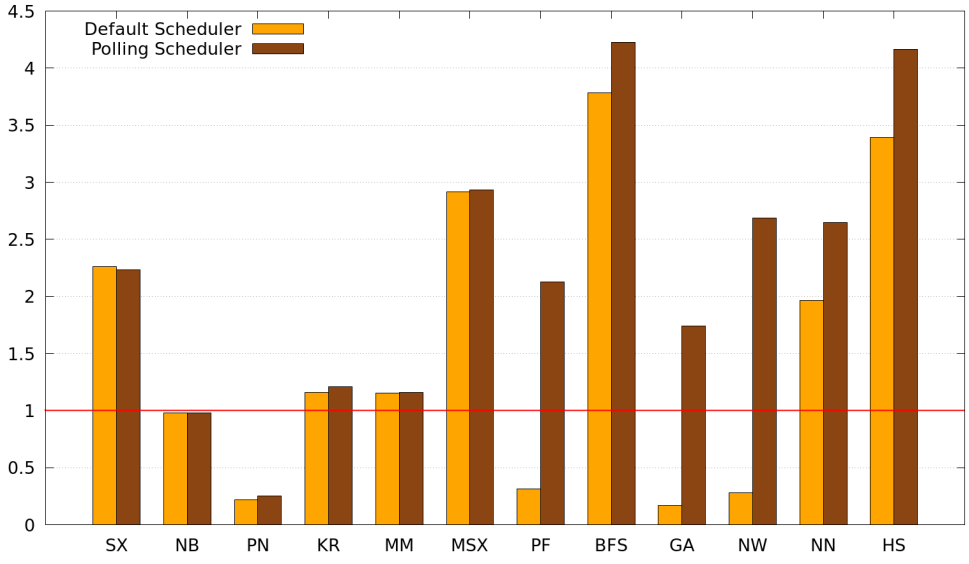

Figure 5.11: Speedup of the different benchmarks using the default and a polling scheduler

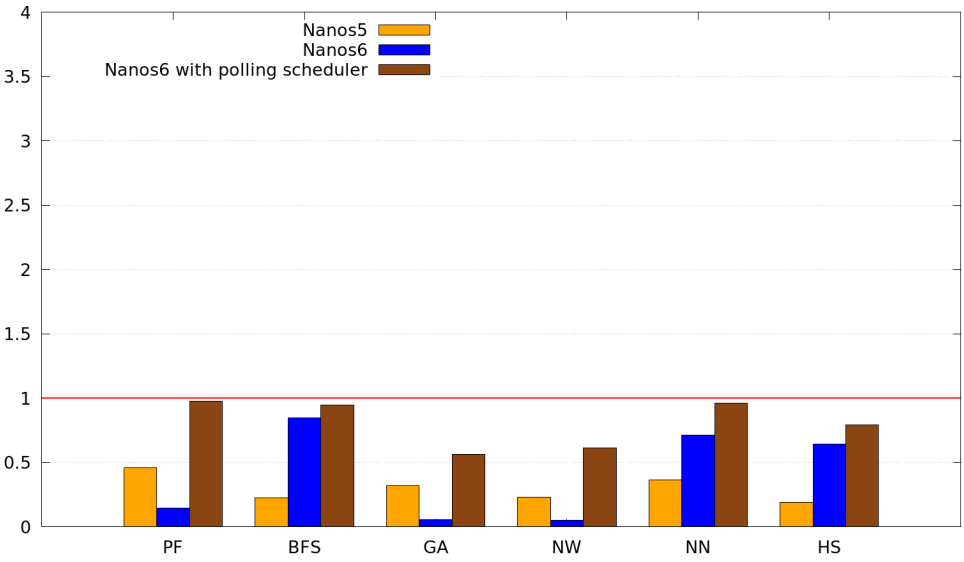

Figure 5.12: Speedup comparison over the original program using Nanos++, Nanos6 and Nanos6 with a polling scheduler

5.3.2 BSC Application Performance

Regarding BSC applications, one of the unexplained results is that of Saxpy and Multi-Saxpy. Both of these applications are very memory intensive, since most of the running time is taken by the memory transferences. In consequence, it would be expected to obtain a slightly worse performance in Nanos6 compared to Nanox due to the usage of unified memory with no optimization functions. The results however, show a speedup in Nanos6.

Analyzing with the visual profiler the traces of these applications, it can be observed that the executions are drastically different in the different OmpSs versions. Comparison pictures of the Saxpy execution are shown in figure 5.13.

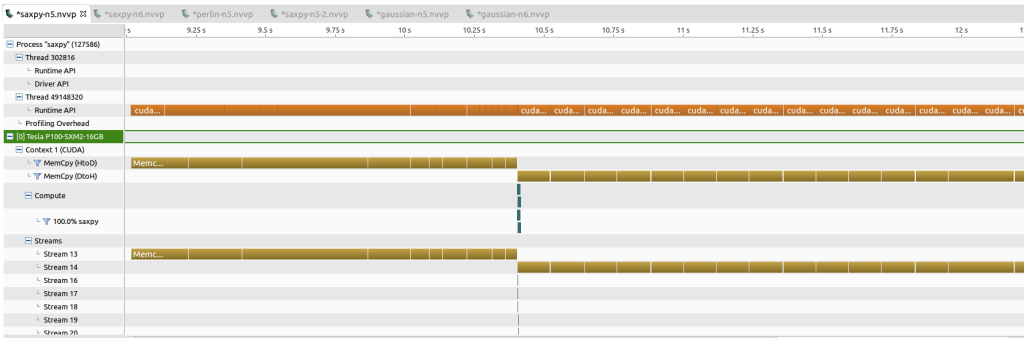

Figure 5.13: Saxpy execution trace

These applications, launch a Saxpy operation divided into several tasks, each processing a portion of the data performing a CPU initialization, kernel computation and CPU correctness checking. In Nanos6, the memory transfers and the kernel executions overlap as much as possible, however, in Nanos++ all the memory operation are executed before any kernel is launched.

Optimally, each kernel would be launched immediately after the host to device copy is performed and the device to host copy would be queued immediately afterwards. Something that can be observed is that the asynchronous memory copies are becoming synchronous, since the operations on the CPU correspond perfectly with the actual transferences, thus, something similar to what happened in Nanos6 with prefetches is happening with memory copies in Nanos5.

This is happening because the warm up runs done on the benchmark before measuring the actual execution are going over the limit of asynchronous operations. In order to avoid this, another measuring of the applications is done without any warm-up, results are shown in figure 5.14. No appreciable difference is found on the results, however, the execution profile of the application is different, as shown in figure 5.15.

In this case the problem is different, now the non overlapping memory operations are the device to host transfers. In this case it is not because calls are becoming synchronous, so no explanation has been found for this behaviour. In the future this issue will be checked with Nanos++ experts in order to avoid a similar shortcoming in Nanos6.

In the case of Perlin Noise, the reasons for bad performance is that big amounts of data are being transferred (2 GB in and out), however, the runtime is overlapping execution and memory operations correctly, thus regular memory copies used by

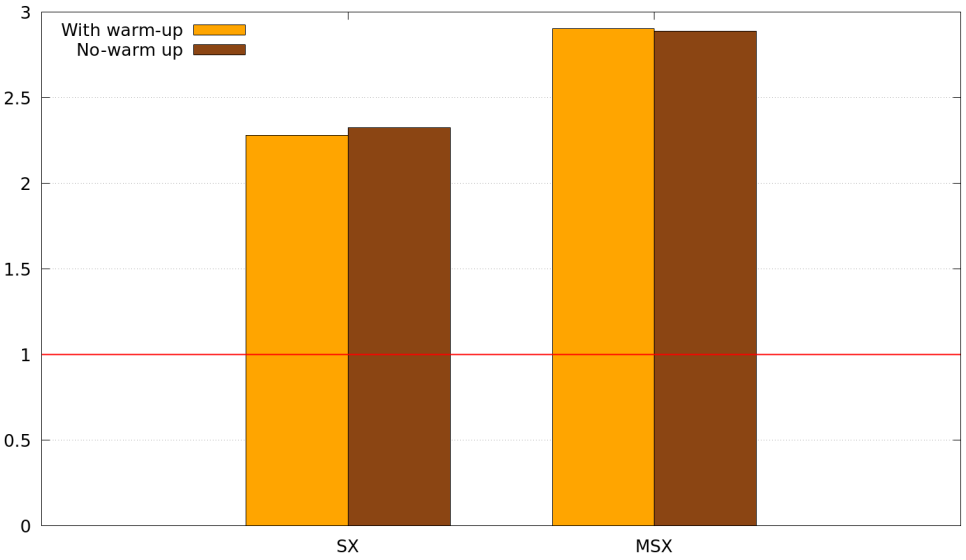

Figure 5.14: Speedup of Saxpy with and without warm-up runs

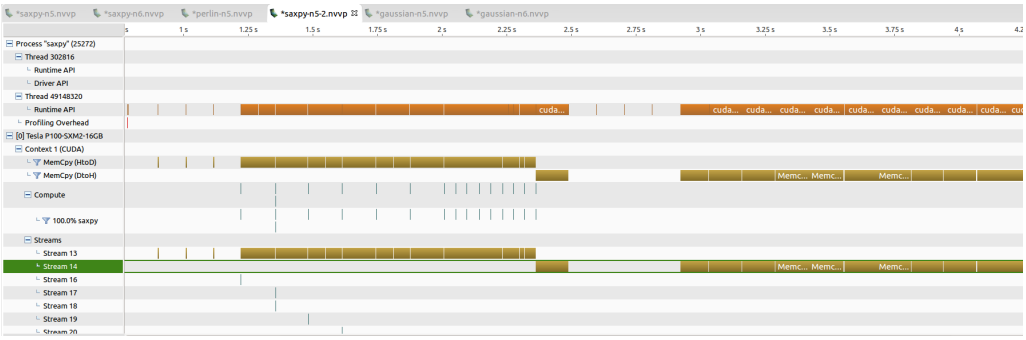

Figure 5.15: Saxpy execution trace with no warm ups

Nanos++ perform better than the unified memory page fault mechanism used by Nanos6.

5.3.3 Common performance considerations

The previous sections have analyzed the performance of some applications with performance issues, however, some explanation is due to the speedups obtained in most applications using Nanos6.

The performance benefits obtained by Nanos6 can be summarized in the following:

- Reduction in host side operations
- Elimination of unnecessary memory transfers

Regarding the reduction in host operations, this comprises several performance issues in Nanos++ that have been solved in the design of Nanos6. First of all there is the necessity to synchronize the memory operations and kernel executions on the host, which introduces unneeded idle time on the GPU. Also, sending all memory operations to two streams causes them to become serialized and introduces false dependencies between tasks that should not have them. Additionally, in order to track data locations, the runtime needs to access and update several shared data structures, which are kept thread safe via locks additionally reducing performance.

The reason why some applications benefited for this while being minimally affected by the low frequency polling issue in Nanos6 is due to the number of concurrent tasks. The tasks affected by the issue all had many non concurrent tasks, however, a program with concurrent tasks can avoid this problem by executing several concurrent kernels while the host is polling for the already finished kernels. The issue becomes less relevant as the time the GPU can spend executing other kernels increases.

Regarding the removal of unnecessary memory transfers, this refers to the fact that Nanos++ copies all the output data on GPU tasks back to the host on synchronization points. Sometimes this is not necessary, for example, the pathfinder application generates data in some kernels which is used as input on others but is never needed on the host, however, the runtime will move it by default since it was specified as output in a task.

Something to consider, however, is that it is possible to introduce unnecessary data movement in Nanos6, specially for programs with a very low memory usage. This can happen because unified memory works on a page faulting basis and copies entire pages. This means that if a kernel touches a page even if a single byte is needed all the page will be copied.

In summary, results are generally positive on the benchmark suite used, due to the improvements made to the compute scheduling side of Nanos6 over the previous version. Memory operation on the other hand, perform sub optimally and future work should be focused on improvement on this aspect of the runtime.

Chapter 6

Conclusions and Future Work

Through the last decade, heterogeneous computing has become a established field on High Performance Computing environments. As HPC hardware advances and becomes more specialized, it presents a challenge to traditional models. Thus OmpSs needs to adapt to more recent paradigms in order to obtain as much performance.

Exploiting the characteristics and features of modern systems is key for the development of the runtime for OmpSs-2, called Nanos6. The goal of the development is to keep the programmability simple while improving the design and performance of the system. For this, the shortcomings of the previous runtime have been analyzed and design work has been done to overcome it.

The usage of unified memory has allowed for a relatively rapid development, as well as avoiding some overheads present on the previous version of the runtime, however, it has introduced a performance penalty on the execution of memory heavy applications.

An number of applications from two sources have been ported to create a benchmarks suite with which to evaluate the application, and with a exception, the results have been generally positive. Some of the issues of the runtime have been exposed, thus future work will be focused in fixing the performance sinks on the system.

In general, the runtime is still in a early version, however, the results are promising, specially considering that no work has yet been done on scheduling policies and multi-GPU computing. While no CUDA supporting version of the runtime is yet released to the public, this first version provides a base in which to build optimizations and additional architecture support for a public release of the runtime, there is however, still much work to do on the runtime.

Future work

One of the first lines of work to consider is, of course, the performance of the runtime. Since good performance has been shown on programs with high computing requirements and low memory usage, a special focus should be put on the execution of high memory using programs. The usage of prefetch operations for the improvement of the memory movements should be revisited in order to study how to overcome its limitations and be used effectively on the system.

Another task to consider is the development and porting of new benchmarks to further evaluate the performance of the runtime. GPU applications are generally divided into compute bound and memory bound applications, depending on the

factor which is most critical to their performance. The evaluation has shown that Nanos6 can perform better than the previous version in applications with small memory transfers and high compute needs, but further analysis is needed for other program types. In addition, it may be necessary to develop some benchmark for collaborative GPU/CPU programs since very few are available and is one of the use cases in which OmpSs has potential to excel.

Considering this, the development of a suite of application to evaluate future versions of OmpSs is foreseeable, in order to develop a process to evaluate the performance of future versions of the runtime, once different optimizations are applied to the runtime. Special emphasis should be put on finding and porting benchmarks with high memory usage. Other sources of applications, such as the CHAI benchmarks are being looked at, in addition to the Rodinia application not yet ported.

In addition, there are still additional features to implement in the Nanos6 CUDA support subsystem, for example, Multi-GPU computing is one of the tasks to work on. Technically it is possible to use multiple GPUs currently, however, since only a naive scheduler is being used and no mechanism exists to keep track of the location of the data on the system, the execution on multiple GPUs would be pseudo-randomly distributed and may cause performance issues due to unnecessary data movements.

For this, work must be done in the scheduling, creating schedulers that can exploit features such as locality.

An additional point to mention in regards to future work is the foreseeable need to develop a subsystem for other heterogeneous programming models. While the current system has been designed considering that other architectures will be added in the future, and many components are general enough to integrate new models easily, a revision of the code base considering adaptability may be desirable.

Bibliography

[1] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming". In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[2] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, pp. 44–60.

[3] Shuai Che et al. "Rodinia: A benchmark suite for heterogeneous computing". In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee. 2009, pp. 44–54.

[4] Eduard Ayguadé et al. "Extending OpenMP to survive the heterogeneous multi-core era". In: *International Journal of Parallel Programming* 38.5-6 (2010), pp. 440–459.

[5] Matthew A Goodrum et al. "Parallelization of particle filter algorithms". In: *International Symposium on Computer Architecture*. Springer. 2010, pp. 139–149.

[6] Jesus Labarta. "Starss: A programming model for the multicore era". In: *PRACE WorkshopNew Languages & Future Technology Prototypes at the Leibniz Supercomputing Centre in Garching (Germany)* 25 (2010).

[7] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[8] H Gelabert and G Sánchez. "Extrae user guide manual for version 2.2. 0". In: *Barcelona Supercomputing Center (B. Sc.)* (2011).

[9] Judit Planas Carbonell. "Programming models and scheduling techniques for heterogeneous architectures". In: (2015).

[10] *cuBLAS :: CUDA Toolkit Documentation*. 2017. URL: `http://docs.nvidia.com/cuda/cublas/` (visited on 04/11/2017).

[11] *CUDA C Programming Guide*. 2017. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide/` (visited on 04/10/2017).

[12] *cuFFT :: CUDA Toolkit Documentation*. 2017. URL: `http://docs.nvidia.com/cuda/cufft/` (visited on 04/11/2017).

[13] Barcelona Supercomputing Center. *Power8+ CTE User Guide*. URL: `https://www.bsc.es/support/POWER_CTE-ug.pdf` (visited on 10/09/2017).

[14] BSC Programming Models department. *BSC Application Repository*. URL: `https://pm.bsc.es/projects/bar` (visited on 10/02/2017).

[15] BSC Programming Models department. *Mercurium — Programming Models @ BSC*. URL: `https://pm.bsc.es/mcxx` (visited on 04/11/2017).

[16] BSC Programming Models department. *Nanos++ — Programming Models @ BSC*. URL: `https://pm.bsc.es/nanox` (visited on 04/11/2017).

[17] BSC Programming Models department. *Ompss Examples and Exercises*. URL: `http://pm.bsc.es/ompss-docs/examples/ompss-ee.tar.gz` (visited on 10/02/2017).